IMPROVING MESSAGE-PASSING PERFORMANCE AND SCALABILITY IN HIGH-PERFORMANCE CLUSTERS

by

Mohammad Javad Rashti

A thesis submitted to the Department of Electrical and Computer Engineering

In conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

(October, 2010)

Abstract

High Performance Computing (HPC) is the key to solving many scientific, financial, and engineering problems. Computer clusters are now the dominant architecture for HPC. The scale of clusters, both in terms of processor per node and the number of nodes, is increasing rapidly, reaching petascales these days and soon to exascales. Inter-process communication plays a significant role in the overall performance of HPC applications. With the continuous enhancements in interconnection technologies and node architectures, the Message Passing Interface (MPI) needs to be improved to effectively utilize the modern technologies for higher performance.

After providing a background, I present a deep analysis of the user level and MPI libraries over modern cluster interconnects: InfiniBand, iWARP Ethernet, and Myrinet. Using novel techniques, I assess characteristics such as overlap and communication progress ability, buffer reuse effect on latency, and multiple-connection scalability. The outcome highlights some of the inefficiencies that exist in the communication libraries.

To improve communication progress and overlap in large message transfers, a method is proposed which uses speculative communication to overlap communication with computation in the MPI Rendezvous protocol. The results show up to 100% communication progress and more than 80% overlap ability over iWARP Ethernet. An adaptation mechanism is employed to avoid overhead on applications that do not benefit from the method due to their timing specifications.

To reduce MPI communication latency, I have proposed a technique that exploits the application buffer reuse characteristics for small messages and eliminates the sender-side copy in both two-sided and one-sided MPI small message transfer protocols. The implementation over InfiniBand improves small message latency up to 20%. The implementation adaptively falls back to the current method if the application does not benefit from the proposed technique.

Finally, to improve scalability of MPI applications on ultra-scale clusters, I have proposed an extension to the current iWARP standard. The extension improves performance and memory usage for large-scale clusters. The extension equips Ethernet with an efficient zero-copy, connection-less datagram transport. The software-level evaluation shows more than 40% performance benefits and 30% memory usage reduction for MPI applications on a 64-core cluster.

Acknowledgements

I start with the name of Almighty God to whom all praise belongs. I greatly appreciate helpful guidelines and continuous support of my supervisor, Professor Ahmad Afsahi, during the development of this dissertation. His meticulous and helpful supervision was a key in conducting this research, without which the completion of this work would not be possible. I would like to acknowledge the financial support from the Natural Science and Engineering Research Council of Canada (NSERC), Ontario Graduate Scholarship (OGS), Nortel Networks Graduate Fellowship, Sun Microsystems of Canada HPCVL Research Scholarship, and Queen's University Student Awards. I am indebted to the Department of Electrical and Computer Engineering for awarding me Teaching Assistantship and Teaching Fellowship.

I would like to thank my friends at the Parallel Processing Research Laboratory, Reza Zamani, Ryan Grant, Jonathan Green, Judicael Zounmevo, and Dr. Ying Qian for their friendly help and support during my Ph.D. period. Special thanks go to Ryan Grant for his active and continuous intellectual and technical collaboration in the datagram iWARP project.

Lastly, I would like to express my special gratitude to my wife, Mahsa, for her continuous and warm support in all ups and downs of my study period. I also warmly appreciate all encouragements and support I received from my family, especially my father.

Table of Contents

List of Figures

List of Tables

Glossary

10GE	10-Gigabit Ethernet
ACK	Acknowledgment
ADI	Abstract Device Interface
API	Application Programming Interface
CEE	Converged Enhanced Ethernet
CI	Channel Interface
CMP	Chip Multi-Processor
CRC	Cyclic Redundancy Code
CTS	Clear To Send
DDP	Direct Data Placement
DDR	Double Data Rate
DMA	Direct Memory Access
FLOPS	Floating-point Operations Per Second
GRH	Global Routing Header
HCA	Host Channel Adapter
HPC	High Performance Computing
IB	InfiniBand

IboE	InfiniBand over Ethernet
IETF	Internet Engineering Task Force
IpoIB	IP over IB
iWARP	Internet Wide Area RDMA Protocol
LAN	Local Area Network
LID	Local ID
LLP	Lower Layer Protocol
MO	Message Offset
MPA	Marker PDU Alignment
MPI	Message Passing Interface
MSN	Message Sequence Number
MTU	Maximum Transfer Unit
MX	Myrinet Express
NFS	Network File System
MxoE	MX over Ethernet
NIC	Network Interface Card
NPB	NAS Parallel Benchmark
NUMA	Non-Uniform Memory Access
OF	OpenFabrics
OFED	OpenFabrics Enterprise Distribution
OS	Operating System
PDU	Protocol Data Unit
PCIe	PCI Express
PIO	Parallel Input/Output

PVFS	Parallel Virtual File System
QDR	Quadruple Data Rate
QN	Queue Number
QoS	Quality of Service
QP	Queue Pair
RC	Reliable Connection
RD	Reliable Datagram
RDMA	Remote Direct Memory Access
RDMAoE	RDMA over Ethernet
RDMAP	RDMA Protocol
RMA	Remote Memory Access
RNIC	RDMA-enabled NIC
RoCEE	RDMA over Converged Enhanced Ethernet
RTR	Ready To Receive
RTS	Ready To Send
SCTP	Stream Control Transmission Protocol
SDP	Socket Direct Protocol
SL	Service Level
SMP	Symmetric Multi-Processor
SRQ	Shared Receive Queue
TCP	Transmission Control Protocol
TLB	Translation Look-aside buffer
TOE	TCP Offload Engine
UC	Unreliable Connection

UD	Unreliable Datagram
UDP	User Datagram Protocol
ULP	Upper Layer Protocol
VPI	Virtual Protocol Interconnect
WAN	Wide Area Network
WR	Work Request
XRC	eXtended Reliable Connection

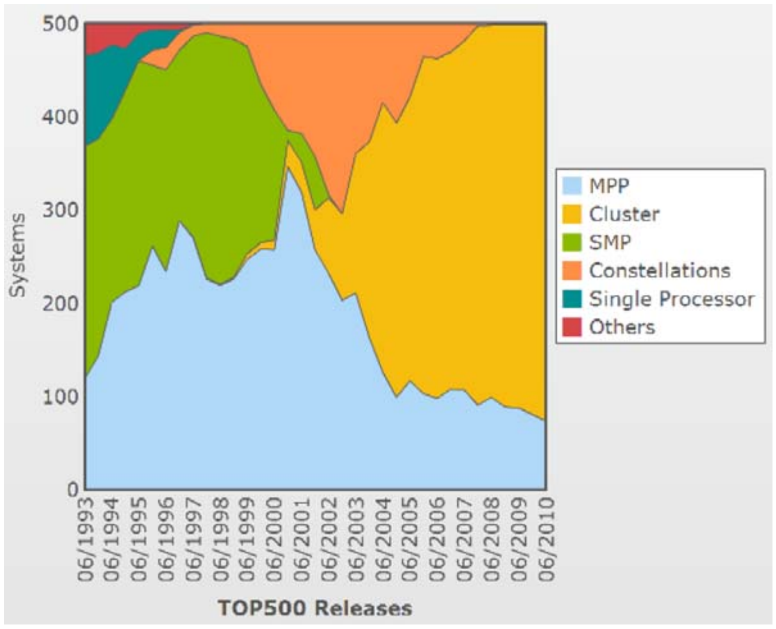
Chapter 1

Introduction

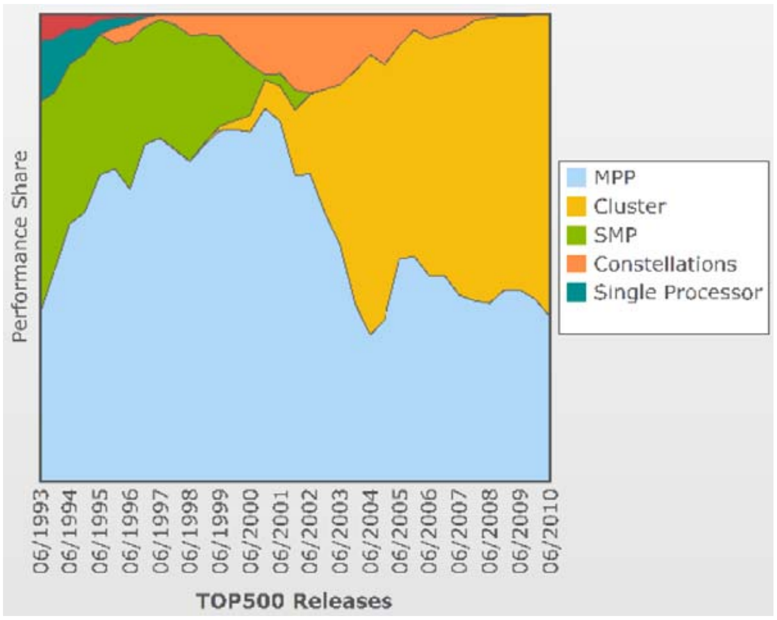
High performance computing or supercomputing involves advanced computing architectures and techniques to deliver superior computational power, efficiency and reliability for challenging applications. HPC is the key to many scientific discoveries and engineering innovations in different areas such as climate science, biology, material science and chemistry, fluid mechanics, and the aviation and automobile industries. In addition, HPC has gained considerable attention in financial management, for applications such as stock market analysis or high frequency trading. For example, currently more than 10% of the top 500 supercomputers are specifically used for financial management [100]. The demand for more computational power and efficiency is continuously increasing in all these areas. Therefore, improving performance of HPC systems is vital for the future of these applications with direct and indirect effects on the quality of our lives.

Computer clusters are highly flexible, configurable, fault-tolerant, and relatively cheap platforms for HPC applications. Modern advancements in computing architectures have made clusters the dominant choice for large scale parallel machines. Multi-core and emerging many-core systems have become the main building blocks for modern clusters. In addition to multi-core architecture, on-board high-performance interconnections (such as HyperTransport [36] and QuickPath [40]), as well as high-speed memory-access mechanisms (such as the recently introduced DDR3 technology) and high-performance node interconnects such as 80Gbps Quad Data Rate (QDR) *InfiniBand* (IB) [39] have enabled clusters to deliver superior computing performance to applications. Consequently, as shown in Figure 1.1 [100], the share of clusters has continuously increased among the top 500 supercomputer sites since year 2000, reaching more than 80% today in terms of the number of systems, and more than 60% in terms of performance.

Nebulae which is currently the second-fastest supercomputer, is a Linux cluster interconnected with an InfiniBand network.



(a)



(b)

Figure 1.1 Share of different architectures among the top 500 supercomputers in terms of (a) the number of systems, (b) performance (reproduced with permission from [100]**)**

Inter-process communication is one of the major requirements of parallel computing. The efficiency of communication significantly affects the performance of parallel applications. Processes of a parallel job can communicate using shared-memory or message-passing models. Due to higher scalability of message-passing systems, this paradigm is currently the dominant parallel programming model in clusters. The *Message Passing Interface* (MPI) [62] is the de facto standard for message-passing in clusters. It should be mentioned that MPI implementations use different means of communication such as shared-memory address space and network to optimize the message-passing performance.

With message-passing clusters being the predominant supercomputing architecture of today, and with the increasing scale of clusters to peta-FLOPS and even exa-FLOPS systems [37] in the next decade, inter-process communication performance will have important application-level performance implications for HPC. There are various approaches for improving communication performance, including latency reduction and hiding, bandwidth increase, and communication scalability improvement. In this dissertation, I have opted to contribute to the performance and scalability of HPC clusters by following some of these approaches to address inefficiencies existing in messaging libraries of modern interconnects.

1.1 Problem Statement

Contemporary interconnection technologies offer advanced features such as *Remote Direct Memory Access* (RDMA) and operating system (OS) bypass, resulting in superior communication performance. With the vast deployment of these interconnects in modern clusters, it is crucial to make sure that clusters can efficiently benefit from the advanced features of these interconnects, and that the raw performance of these interconnects can translate into application performance. This is particularly more important with the emergence of large-scale multi-core and many-core clusters, which put substantially more pressure on the communication processing engines. To achieve such goals, it is therefore important to improve the efficiency and scalability of communication libraries that directly work with the computing and communication hardware in a cluster.

In this research, I address the following questions:

- How can current and future parallel computing architectures benefit from using modern interconnection networks? How do existing message-passing libraries take advantage of these advanced technologies?
- What are the inefficiencies in communication libraries that prevent them from achieving low communication latencies offered by modern interconnects?
- How can we improve the potential for making progress in communication and for overlapping communication with computation in order to effectively hide the communication latency?
- How can we make communication libraries more scalable and resource-efficient for future peta-FLOPS and exa-FLOPS clusters?

To answer these questions, the work is directed towards understanding state-of-the-art cluster interconnection technologies and the message-passing libraries that use them. In particular, this research is focused on the MPI message-passing library, running on top of user level communication interfaces. The approach taken is to assess the performance and scalability of modern messaging libraries and explore opportunities to reduce or hide communication latency and to increase communication bandwidth and scalability for these libraries. Different techniques such as copy avoidance, speculative communication, and use of low-overhead transport protocols are utilized to achieve these goals.

1.2 Dissertation Contributions

In this section, I present the outcomes of my research in addressing some sources of inefficiency in message-passing over modern HPC clusters to improve MPI performance and scalability.

1.2.1 Messaging Layer Performance and Scalability Analysis

In Chapter 3 of the dissertation, I present a deep assessment and analysis of the user-level communication libraries as well as MPI over high-performance cluster interconnects. The outcome of this work, presented in Chapter 3, highlights some of the inefficiencies that exist in these libraries. In particular, this part of the work uses novel techniques to assess deep performance characteristics of the interconnects and their respective libraries [78].

I have measured the ability of the messaging libraries in making independent progress in communication and overlapping the communication with computation in order to hide the latency [82]. The investigations confirm that some inefficiencies in the libraries exist that prevent independent progress and consequently reduce the overlap ability.

I have also confirmed another source of latency overhead to be the memory registration and memory copies in small data transfers. Frequent buffer reuse may help improve the latency for large messages, but it does not affect the small-message transfers much due to the way the small-message transfer protocols are designed [78].

The research also shows the scalability of modern interconnects for multi-core clusters to some extent. However, there are alerting signs for future many-core systems. The results show that network interface cards (NICs) stop scaling in performance when eight or more cores are active per node [78, 80].

1.2.2 Independent Progress and Latency Hiding for Large-Message Transfer

In this part of the dissertation presented in Chapter 4, I have proposed an innovative MPI Rendezvous protocol on top of RDMA-enabled networks to increase the chance of communication progress and overlap in order to better hide the communication latency. The proposed method speculatively starts communication from the receiver side, if it arrives earlier than the sender. The protocol is adaptive, meaning that it is designed to fall back to the older Rendezvous protocol, if due to application timing the speculation incurs more overhead on the performance than it brings benefits. Micro-benchmark results show that the proposal increases the communication progress ability to 100% and the overlap ability to more than 80%. The improvements are achieved in communication scenarios that the old Rendezvous protocol shows almost no communication progress and overlap ability [83]. The results also show up to 30% reduction in communication wait time for some applications when non-blocking calls are used [81].

1.2.3 Small-Message Latency Reduction for Frequently-used Buffers

This part of the dissertation presented in Chapter 5 is aimed at reducing small-message latency in MPI applications. The improvement is accomplished by proposing changes to different MPI small-message transfer protocols for frequently-used buffers. The method keeps track of buffer usage in the application and bypasses one buffer copy at the sender side when it finds out that a buffer is being used frequently.

The protocol is adaptive and falls back to the corresponding regular protocol if the buffer usage frequency often falls below a pre-determined threshold. Up to 20% latency reduction is achieved by using the proposed method for small messages on InfiniBand network. The method also shows considerable communication time improvement as well as runtime reduction for some MPI applications [77, 84].

1.2.4 Extending the iWARP Ethernet Standard to the Datagram Domain

To increase the scalability of iWARP-based HPC clusters, Chapter 6 of this dissertation presents a proposal on extending the iWARP standard over datagram transports, and in particular, *User Datagram Protocol* (UDP). The main contributions of this part are the proposed extensions to the standard and the guidelines to modify different layers of the standard to support datagram transport. The other contributions include design and implementation of the proposed datagram iWARP in software, the *OpenFabrics* (OF) verbs user-level interface on top of the software-based datagram iWARP and adapting the MPI library implementation to make use of the datagram iWARP [79].

The evaluation results show substantial reduction in the memory footprint of HPC applications using MPI on top of the proposed datagram iWARP. The results on a 64-core cluster show up to 30% decrease in total application memory footprint. The results also show high scalability in memory savings. This implies that considerably higher amounts of memory savings are expected for larger clusters.

The results of this work also affirm substantial performance improvement for MPI applications. Communication time and application runtime are decreased by up to 45% on a 64-core cluster. Similarly, higher performance benefits are expected on larger machines.

1.3 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 provides background on HPC clusters, MPI, interconnection networks and their messaging libraries. Chapter 3 presents an evaluation and analysis of modern cluster interconnects, and MPI implementations on top of them. In Chapter 4, a speculative and adaptive Rendezvous protocol is presented for improving large message transfers using MPI. In Chapter 5, an adaptive technique is presented to reduce communication latency of MPI small-message transfer protocols. In Chapter 6, to address scalability concerns of MPI applications on large-scale iWARP clusters, an extension is proposed to the iWARP Ethernet protocol to support connection-less datagram transport. Chapter 7 provides conclusions and outlines future research directions.

Chapter 2

Background

Commodity clusters are the dominant platform for HPC applications and data centers, mostly due to their low price of ownership, high performance, and availability. This chapter provides background on cluster architecture with a focus on interconnection networks and messaging libraries. Specifically, I will discuss message-passing communication subsystem and in particular, the MPI standard, MPI implementations, interconnection networks, and their user-level libraries.

2.1 Computer Clusters

A computer cluster is a network of independent computing nodes, utilized to run a set of parallel jobs. Figure 2.1 shows a general architecture for a cluster. The standalone nodes can be commodity PCs/workstations, *Symmetric Multiprocessor* (SMP) nodes, *Non-Uniform Memory Access Architectures* (NUMA), or multi-core systems, each running an independent copy of the OS. Cluster nodes are connected via interconnection networks. A cluster interconnect can be a high-performance cluster-optimized network or a commodity network such as traditional Ethernet. Modern high-performance networks bypass the OS and remove several sources of communication latency. Due to frequent inter-node communications in HPC applications, HPC clusters benefit consistently from high-bandwidth, low-latency interconnects, despite the relatively higher cost associated with them.

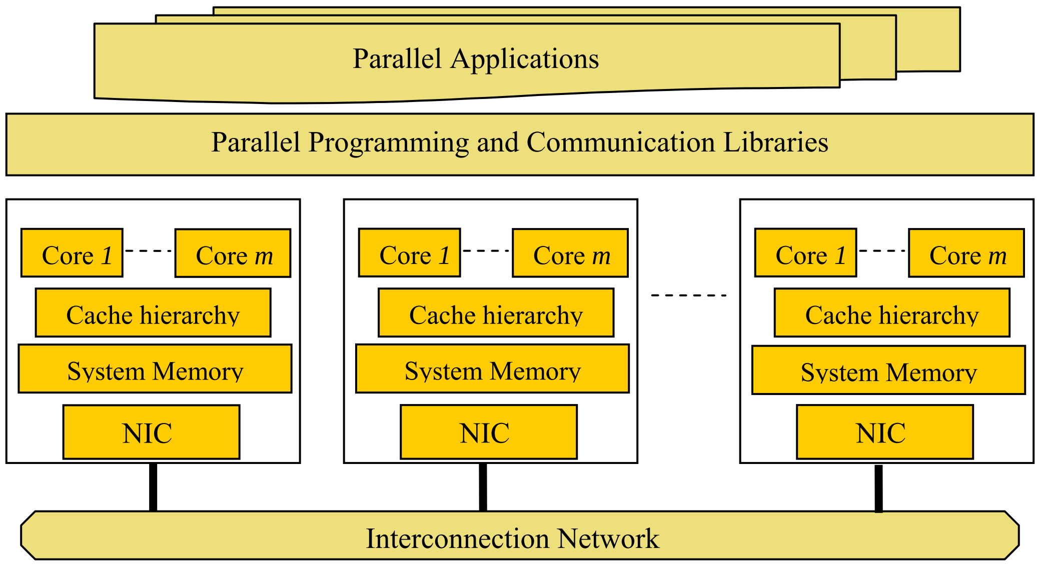


Figure 2.1 A general view of a computer cluster architecture

2.2 Message-Passing Communication Subsystems

Due to its high scalability, the message-passing model is the major parallel programming paradigm in clusters. In message-passing clusters, the communication subsystem includes interconnection networks and message-passing communication libraries.

Figure 2.2 shows a schematic of the communication subsystem architecture in a message-passing cluster. At the top level, below the user application, we have high level messaging libraries such as MPI or Charm++ [74]. These libraries reside either on top of kernel space communication primitives such as the socket application programming interface (API), or over the user-level communication libraries that bypass the OS.

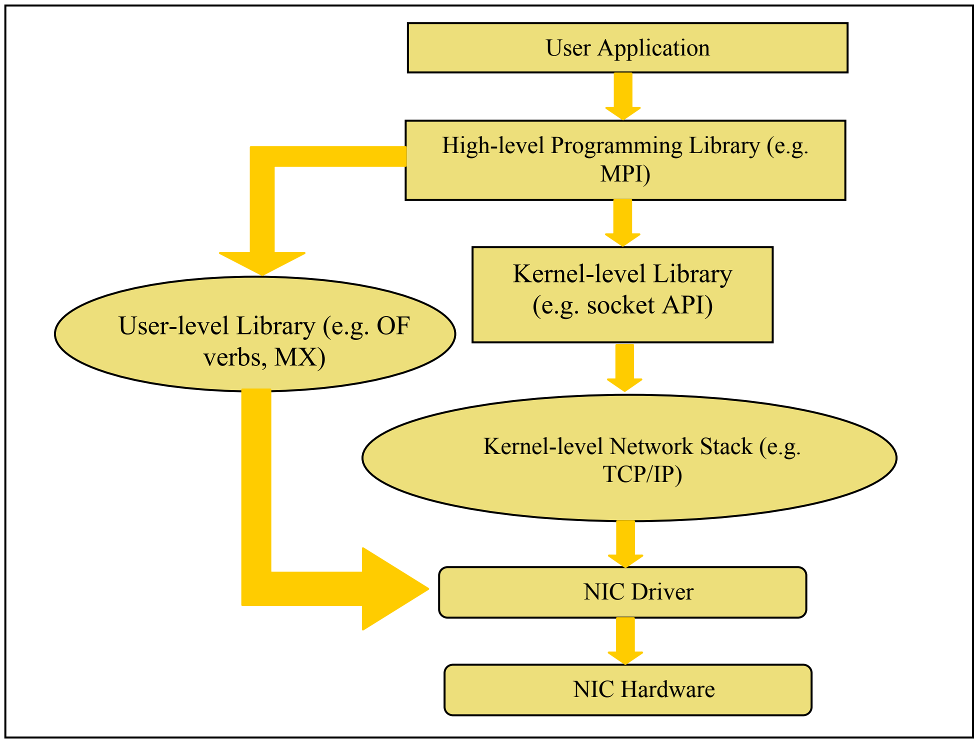

User Application
High-level Programming Library (e.g. MPI)
Kernel-level Library (e.g. socket API)
User-level Library (e.g. OF verbs, MX)
Kernel-level Network Stack (e.g. TCP/IP)
NIC Driver
NIC Hardware

Figure 2.2 Messaging libraries at different layers

2.2.1 Kernel Level Messaging Libraries

Traditional communication libraries reside in the kernel space. Kernel-level libraries are mostly utilized for datacenter services and internetworking. These libraries such as the socket API use kernel level network stacks (e.g. the TCP/IP stack) to access the network. The UNIX-style socket programming API defines a client/server model of communication over the TCP/IP protocol stack. Reliable, in-order communication and flow/congestion control are basic overheads of a TCP connection. Connectionless socket programming is also possible using unreliable datagram sockets which use UDP instead of TCP. TCP/IP uses message buffering and copying in several layers of the protocol, which imposes a large overhead on performing message transfers. When using faster interconnection hardware, the host protocol-processing overhead becomes the dominant part of the total communication latency, making TCP/IP and its socket API inappropriate for high-performance applications. Protocol overhead includes reliability and control services, multi-layer data copy and encapsulation, multiple buffer copies, and kernel-layer API call overheads.

2.2.2 User-Level Messaging Libraries

To simplify the protocol stack and decrease its processing overhead, *user-level messaging libraries* are introduced to reside between the high level programming libraries (such as MPI) and the NIC driver modules. These libraries avoid system calls and provide a low-level, hardware-specific communication API. They also reduce the number of intermediate message copies. This is done by eliminating kernel buffer copies and exploiting RDMA techniques to directly copy the data into the final destination buffer, with the assistance of NIC offload engines. Examples of user-level libraries are *Myrinet Express* (MX) for Myrinet [65] and OpenFabrics verbs for iWARP and InfiniBand [71]. The following sections provide background on MPI, interconnection networks, and their user-level messaging libraries used throughout the dissertation.

2.3 Message Passing Interface (MPI)

MPI [63] is a message-passing standard proposed in 1993 by MPI forum for loosely-coupled parallel machines. The standard provides a set of portable messaging interfaces that could be implemented on any machine. The main skeleton of the standard consists of: point-to-point communication, collective communication, one-sided communication, process groups, communication contexts, process topologies, language bindings for C and Fortran, environmental management, and profiling interface [63].

Currently, there are two MPI specifications: MPI-1 and MPI-2. MPI-2 was introduced in 1997, and the latest extensions were made in 2009 as MPI 2.2. The MPI Forum is currently making revisions to the standard and finalizing the next generation of MPI known as MPI-3 [62].

Because MPI is the de facto standard for message-passing, I will use it as the main communication standard in the evaluation of modern interconnects throughout my research. In this section, I will provide a brief introduction to the MPI standard. The next section will discuss MPI implementations.

2.3.1 Point-to-point Communications

The basic communication model in MPI is *point-to-point communication*, in which two sides are involved. The basic communication primitives in this model are the point-to-point *send* and *receive* functions. The sender should call a *send* routine and the receiver needs to call a *receive* routine. These calls may block the calling process (*blocking mode*) until it is safe to use the communication buffer again [63]. At the sender side, this may happen as soon as the message is delivered to the MPI layer of the sender node. The *blocking receive* blocks the receiver process until the receive operation is completed.

There are *non-blocking* alternatives for MPI send and receive operations. A non-blocking send may return control to the application as soon as the sender process invokes the library to make progress in the communication procedure. A non-blocking receive also returns immediately after posting the receive request to the MPI level, leaving it to the messaging library to advance the communication. Both sender and receiver should wait or poll for the completion of the operation, by calling an appropriate library routine such as MPI *wait* or MPI *test*.

MPI implementations use a *progress engine* to manage the progress of communication after a non-blocking call returns. *Independent progress* means that a pending non-blocking communication request makes progress without requiring any future library calls. This independent progress can be achieved with the assistance of a hardware unit at the NIC, or by a helper thread. Supporting independent progress depends on factors such as: the communication protocols used in the communication library, the form of MPI progress engine (synchronous or asynchronous), and the offloading support to the NIC or onloading support to the available cores.

Figure 2.3 illustrates an example of implementing non-blocking communications with and without hardware assistance. In this figure, the dark bars in part (a) show the host CPU involved in communication.

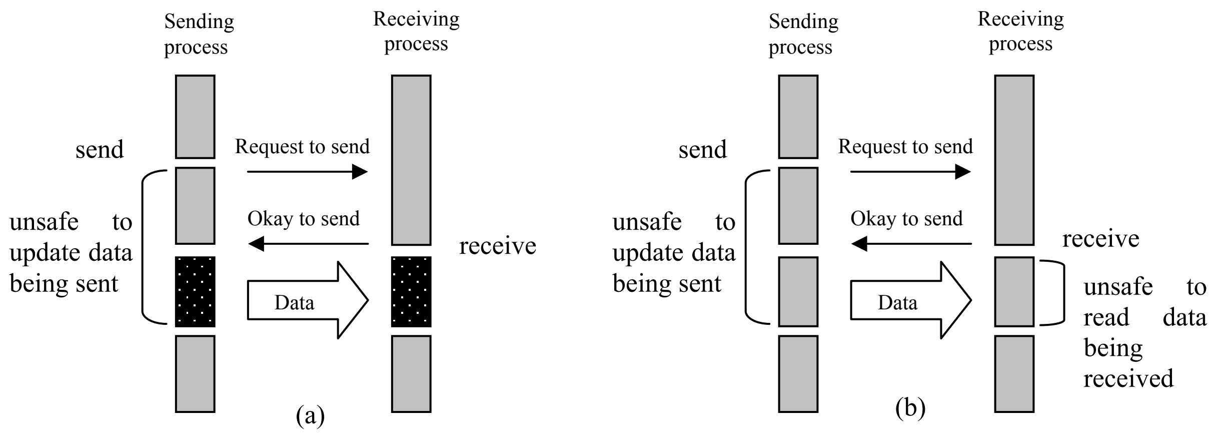


Figure 2.3 Non-blocking communication (a) without NIC hardware assistance, (b) with NIC hardware assistance (adapted from [28]**)**

2.3.2 Collective Communications

Other than point-to-point communications, MPI supports collective communications that involve a group of processes. Collective operations can be classified in three groups: synchronization operations (such as *barrier*), data movement operations (such as *broadcast, gather, scatter,* and *all-gather*) and computation operations (such as *reduce, all-reduce,* and *scan*) [63]. MPI collective operations are currently defined as blocking operations, which means that the calling process blocks until the operation is complete. However, a proposal has been made for MPI-3 specification to include non-blocking collective operations [32].

2.3.3 One-sided Communications

Other than the two-sided communication model, there is another model of message-passing called *one-sided communication*, where only one of the communicating processes is engaged in data transfer. This model, conceptually depicted in Figure 2.4, is based on the *Remote Memory Access* (RMA) model, which is supported in MPI-2 standard. The *origin* process gains access to a portion (window) of the memory address space of the *target* process and can directly *put* or *get* data without engaging the other process. In addition to put and get, the MPI one-sided operations include *atomic* memory operations such as *accumulate*. This kind of direct memory access is abstracted from any real underlying shared-memory facility. Therefore, MPI-2 can perform such operations even without any hardware assistance for shared-memory [53]. However, one-sided operations give much of shared-memory flexibility and if assisted in hardware, the can improve message-passing performance.

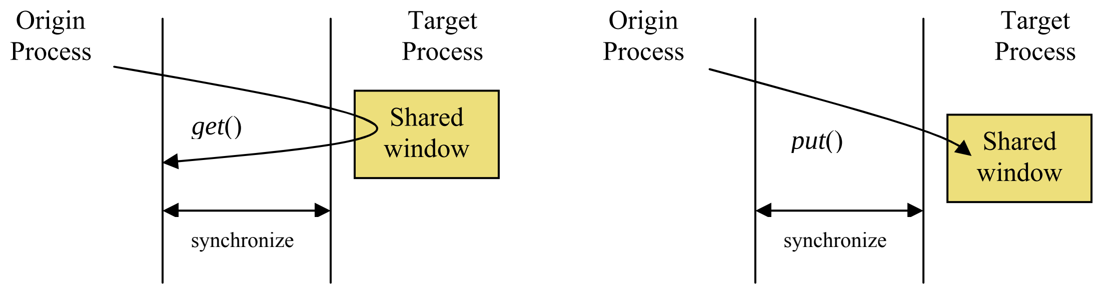


Figure 2.4 One-sided get and put operations

The above approach can utilize underlying hardware or available extra cores on the node, to avoid intermediate buffering at the receiver side, making it possible to achieve zero-copy communication [99]. However, explicit synchronization is needed to prevent access races. If the target process is explicitly involved in synchronization, the one-sided communication mode is called *active target*. This is similar to two-sided communication, except that all transfer arguments including buffer addresses are provided by one process. The other mode where the target process is not involved is called *passive target*. MPI has three models of one-sided synchronization, which are summarized below.

- *Fence*: As a means of active target synchronization, MPI_Fence is a collective operation which blocks the caller processes until all one-sided operations on the corresponding window are complete.
- *Post-wait and Start-complete*: The target posts the window to make it ready for origin processes. Then, it waits for completion of one-sided operations performed by origins. Each origin process starts the window to be able to launch one-sided data transfers. By calling the complete call the origin waits to make sure that all pending one-sided operations on the corresponding window are complete. These calls are used for active target synchronization.
- *Lock-unlock*: To make passive target synchronization possible, the lock operation gives the calling process exclusive access to the window or it blocks that process if the exclusive access cannot be granted. Unlock waits for completion of one-sided operations on the window and makes the window ready for future operations. In this case, the target process is not involved in synchronization.

2.3.4 Communication Contexts and Communicators

MPI provides *communication contexts* to modulate and separate creation of distinct message streams without confusion. For example, calling two separate functions for sending/receiving messages in the same program will not confuse the communication of messages, because of the existence of contexts. The MPI contexts are opaque and the user never manipulates them [63].

MPI also introduces the concept of *process groups* that refers to a set of communicating processes. Each process in a group has a unique process rank. The scope of the communication is defined by the abstract concept of a *communicator*. A communicator refers to one or more process groups and communication contexts. Two types of communicators exist. An *intra-communicator* used for communication between processes of a single group. An *inter-communicator* is used for communication between processes of two groups, one of which is local and the other one is remote [63].

Each message in MPI has an envelope that is used to distinguish messages and selectively receive them. The message envelope includes the source rank, destination rank, tag and communicator [63].

2.4 MPI Implementations

There are several commercial and free implementations of the MPI standard. Free and open-source implementations are popular, especially in academia. They usually support several interconnection networks and operating systems. The open-source MPI implementations include MPICH, MVAPICH, and OpenMPI [70]. Commercial implementations include Intel MPI, Platform MPI and Microsoft MPI. In this section, I briefly explain MPICH, MPICH over Myrinet and MVAPICH, which are used throughout the dissertation.

2.4.1 MPICH Implementation

MPICH is provided by Argonne National Laboratory and is one of the most popular open source implementations [3]. It was intended to be a portable implementation of MPI, and therefore it has been optimized mainly for portability. It supports clusters with various operating systems and hardware architectures. MPICH-2 [4] is the MPI-2 implementation of MPICH and replaces MPICH-1 that was the implementation of the MPI-1 standard.

In terms of software architecture, MPICH has an abstract interface layer for implementation of API routines, opaque objects, data types, and many other parts of the API. This layer, which is called the *Abstract Device Interface* (ADI) is abstracted from the underlying hardware and OS and is portable. ADI3 is the latest version of ADI, providing routines to support both MPI-1 and MPI-2.

The lowest layer in MPICH is called the *Channel Interface* or CI. It is used to implement the ADI with several models of communication for different architectures. Some examples of the MPICH channels are *sock* (implemented with sockets) and *ch_smp* (for intra-node communication). Figure 2.5 shows a general view of the MPICH software architecture. The MPICH2 CH3 device is the intermediate layer that integrates all hardware specific channel implementations into a common interface to be used by MPICH-2 ADI3. The most recent communication channel in the MPICH2 implementation is the *Nemesis* communication subsystem [13]. Nemesis is a channel primarily designed for scalable communication and high-performance intra-node communications.

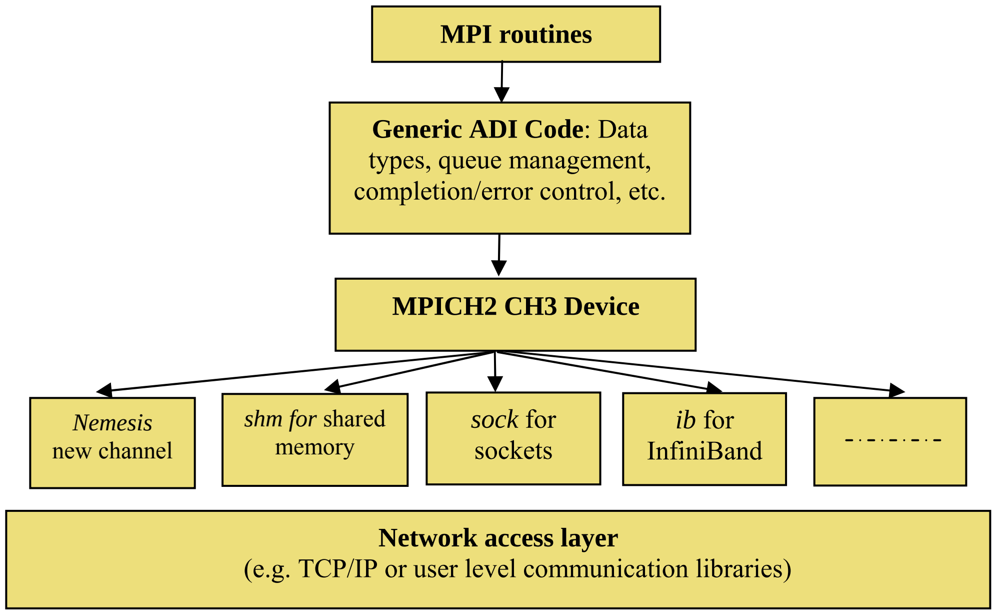


Figure 2.5 MPICH software architecture

Point-to-point and One-sided Communications

MPICH treats small and large messages differently. MPICH implements two different protocols. The *Eager* protocol is used for small messages in point-to-point communications. In this protocol, the sender transfers the message to the receiver, even before a matching receive call is posted and a specific buffer is allocated. If a receive call is not posted yet, the receiver buffers the message as an *unexpected message* in an *unexpected queue* (Unexq). Temporary buffering in the Eager protocol can consume a large amount of memory, depending on the scale of the cluster.

MPICH uses the Rendezvous protocol for large-message communication. The Rendezvous protocol, starts the negotiation when a sender has a message for transfer, and waits for the receiver to become ready for reception. At this point, the actual message data will be transmitted. MPICH also implements *Put* and *Get* protocols for direct transfer of data using a one-sided (RMA) model. Similar to two-sided protocols, small and large messages are treated differently in one-sided put and get protocols.

Intra-node Communication

Parallel processes can run on a single node such as an SMP, NUMA, or multi-core (*Chip Multi-Processor* or CMP) systems. MPICH recognizes intra-node communication and performs local memory transfers. Some intra-node message-passing methods are discussed in [14] and [13], as listed below.

- *User-space memory copy*: A shared-memory region between two processes can be managed as an intermediate space for message exchange. Such a method will suffer from extra memory copies and increased cache pollution. However, there is no kernel involvement through system call or communication stack traversal.
- *NIC-based loop-back*: Almost all network cards distinguish intra-node messages and loop-back the data to the same node. No network transfer is performed in this method, but the data have to traverse the entire network stack in addition to the I/O bus. On the other hand, in modern networks that offload parts of (or the entire) communication stack, this method has the least number of memory copies, leading to lower CPU utilization and reduced cache pollution, especially for large messages.

- *Kernel-assisted memory mapping*: Using kernel assistance, a process can map parts of its memory space into the kernel's memory space. Then, these parts can be accessed by the other processes using kernel modules. Although such a method suffers from kernel/user space switching overhead and is not portable among different operating systems, it eliminates memory copies. As a result, latency is reduced and CPU availability is increased, which can enhance the potential for communication/computation overlap.
- *Queue-based model*: Intra-node communication in MPICH2 Nemesis channel is performed by using lock-free queues in shared memory. Each process has a single receive queue to poll, rather than having multiple receive queues to poll, one for each remote process. The sending process dequeues a queue element from the free queue, fills the queue element with the message, and enqueues it on the receiving process's receive queue. The receiving process then dequeues the element from its receive queue, handles the message, and enqueues the queue element back onto the same free queue where it had originally been dequeued [13].

2.4.2 MVAPICH Implementation

MVAPICH [68] is the high-performance open-source implementation of MPI over InfiniBand, iWARP, and *RDMA over Ethernet* (RDMAoE). MVAPICH is based on MPICH and has its own channels to integrate the MPI over OpenFabrics user-level library [71], which is used for modern interconnects such as InfiniBand and iWARP. MVAPICH2 is the MPI-2 implementation of MVAPICH. MVAPICH and MVAPICH2 are distinct implementations because MVAPICH2 is based on MPICH2 whereas MVAPICH is based on the older MPICH-1 implementation.

The MVAPICH family is provided and maintained by the Network-Based Computing Laboratory at Ohio State University [68]. MVAPICH and MVAPICH2 are continuously updated with support for the most recent advanced features in modern interconnects such as InfiniBand and iWARP. MVAPICH implementation family support advanced features, which include [68]:

- on-demand connection establishment for reduced memory usage
- UD-based and hybrid RC/UD-based channels for scalable and adaptive communication
- enhanced RDMA-based collective operations
- use of *Shared Receive Queues* (SRQ) and *eXtended Reliable Connection* (XRC) for memory scalability
- multi-core aware intra-node communications
- multi-port and multi-rail communication support
- enhanced Rendezvous protocol for increased overlap

2.4.3 MPICH-GM and MPICH-MX over Myrinet

Myricom provides MPI implementations for Myrinet GM and MX libraries [64]. Each of them implements the MPICH channel interface for the corresponding user-level library (either GM or MX) over the Myrinet network [64]. MX is a library with semantics close to MPI. Therefore, the implementation of MPICH over MX is relatively simple and has low overhead. MPICH2-MX is the MPI-2 version of MPICH-MX, but it does not implement one-sided operations, dynamic process management, or multi-threading features of MPI [64]. To utilize such features of MPI over Myrinet, MPICH2 over Nemesis channel [13] can be used.

2.5 Modern Interconnects and their User level Libraries

Due to the significant role that inter-process communication plays in the performance of computer clusters, network performance has a considerable impact on the overall performance of the HPC applications running on clusters. Today, several high-speed interconnection networks are being used in high-performance clusters. Ethernet is the predominant network used in modern clusters. However, the host-based implementation of TCP/IP over Ethernet incurs high communication latency and high host CPU utilization. This has been magnified with the introduction of 10-Gigabit Ethernet (10GE) because it increases the load of communication processing on the host. These shortcomings are not acceptable for today's CPU and memory hungry high-performance applications.

Two trends have driven efforts to overcome Ethernet's inadequacies: accelerating communication stacks over Ethernet, and introducing cluster-specific interconnection networks with their advanced features.

In accelerating Ethernet-based communication processing, the RDMA consortium [85] and the Internet Engineering Task Force (IETF) [38] have developed a set of standard extensions to TCP/IP and Ethernet to decrease host-based overheads. These specifications are collectively known as the *Internet Wide Area RDMA Protocol* (iWARP), which will be described in more detail in Section 2.5.3. Alternatively, cluster-optimized high-performance interconnects such as Myrinet [64] and InfiniBand [39] with fast communication processing engines have been introduced. Such interconnects are the backbone for scalable, high-performance clusters by offering low-latency and high-bandwidth communication.

Modern interconnection technologies are tightly coupled with modern node architectures to offer an efficient utilization of the available architectural technologies on the node. Several techniques are used in modern interconnects to improve performance and remove various sources of overhead in communication.

- Avoiding extra buffer copies using Direct Memory Access (DMA) technologies: This includes *Remote DMA (RDMA)* technology. As shown in Figure 2.6 RDMA (illustrated in light-color arrows) is a one-sided operation, where a node has access to the memory of a remote node. In contrast to the regular host based communication (shown in dark-color arrows), it allows direct data transfer from the source buffer to the destination buffer at the remote node without the host processor intervention or any intermediate CPU-based copy. Conceptually, RDMA has the same meaning as MPI RMA model with the difference that RMA is an abstract concept at the higher layer, while RDMA represents a hardware level technology.

- Bypassing OS overheads such as kernel locks and OS space copies: Use-level libraries are introduced to directly give hardware access to user space, without passing through OS barriers.
- Offloading some parts of communication overhead from the host processor to leave more processing power for computation.
- Onloading some parts of communication processing on available processor cycles to utilize the processing power of multi-core and many-core systems: This technique which seems contrary to the offloading method is more suitable for emerging many-core systems and can be combined with the offloading as well [47].

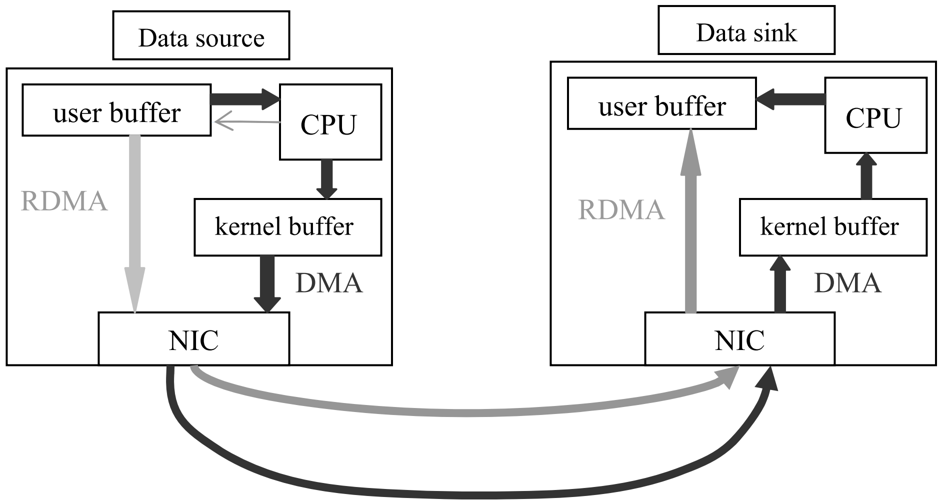


Figure 2.6 RDMA technology mechanism

In the following sub-sections, I will provide an overview of the state-of-the-art cluster interconnection networks and their messaging libraries.

2.5.1 Myrinet

As one of the earliest high performance cluster networks, Myrinet was first introduced in 1994 by Myricom [64]. Myrinet is currently used in less than 2% of the world's top 500 supercomputers. The latest product of Myrinet is Myri-10G, which is actually a 10-gigabit network. Myrinet enables low-overhead Ethernet emulation at link speed (to use the NIC in a Gigabit Ethernet network) and the NICs can work in two network modes: Myri-10G and 10-Gigabit Ethernet. Myri-10G NICs offload some stateless network processing such as checksum calculations, segmentation and re-assembly. However, they are not full *TCP Offload Engines* (TOEs). The user can take advantage of libraries that bypass the kernel (such as MX [65]), sockets (Sockets-MX) [64] and MPI. The NIC is installed on *PCI Express* (PCIe) on-board interconnect [8]. The bandwidth can reach 1.25Gb/s in each direction, for single lane (x1) PCI-Express.

MX is a high-performance, low-level, message-passing software interface tailored for Myrinet. The MX-10G library has semantics close to MPI. The basic communication primitives of MX-10G are non-blocking send and receive operations that are directly used in the implementation of MPI communication primitives. Using this library, initiation of any communication is separated from its completion, to enable communication/computation overlap. Receive completion has an in-order reliable matching operation to make sure that messages are matched in order, based on posted receive calls. MX avoids explicit memory registration. However, an internal registration mechanism with a user-enabled registration cache exists. The library pipelines messages on the physical link for increased bandwidth.

The MX library uses three communication channels: the network channel where messages are transferred through the NIC and the network, the shared-memory channel where messages are exchanged between processes on the same machine, and the self channel for intra-process message that is optimized for messages sent by a process to itself [65]. The use of these channels for different communication modes (i.e. inter-node, intra-node, and self) is regulated by some environment variables. However, intra-node communication uses the shared-memory channel by default.

MX currently does not provide one-sided and collective operations, but it is intended to be available in future releases. Using MX-10G, nodes can run MPI applications through a Myrinet switch (MX-10G over Myrinet or MXoM) or through a 10GE switch (MX-10G over Ethernet or MXoE).

2.5.2 InfiniBand

InfiniBand is an I/O interconnection technology consisting of end nodes and switches managed by a central subnet manager [39]. End nodes use *Host Channel Adapters* (HCA) to connect to the network.

The InfiniBand architecture released by the InfiniBand Trade Association (IBTA) specifies the abstract architectural and functional requirements for InfiniBand HCAs. The user-level messaging standard over InfiniBand consists of some abstract definitions called InfiniBand *verbs* that provide a software interface for programming over the HCA [39]. InfiniBand verbs form the lowest level of software to access the IB protocol-processing engine offloaded to the HCA or onloaded to other processing cores. IB verbs support four transport models for communication [39]. *Reliable Connection* (RC) is a connection-oriented service with guaranteed delivery. *Unreliable Connection* (UC) is a connection-oriented but unreliable service. *Reliable Datagram* (RD) is a connection-less service with guaranteed delivery – it is currently not available in hardware. Finally, *Unreliable Datagram* (UD) is a connection-less and unreliable service.

As shown in Figure 2.7, the verbs layer has semantics based on the *Queue Pair* (QP) model, in which processes post send or receive *Work Requests* (WR) to available send or receive queues, respectively. Queue pairs are able to work in any of the four IB transport models. The WR is used by the user to send data from the source process address space into the send queue, or to wait for matching data to arrive into the receive queue. A *Completion Queue* (CQ) associated with the QP is used by the hardware to place completion notifications for each WR. IB verbs require registration of memory buffers prior to their use.

InfiniBand supports two models of communication semantics. The c*hannel semantics* model is supported by traditional two-sided send and receive operations. In this model, a receive request is needed to be pre-posted before a message arrives. This is required to make sure that the arriving message can be directly placed in the remote memory while the memory address does not need to be exposed remotely. The m*emory semantics* model uses one-sided RDMA operations, for which the remote memory address is exposed and can be directly accessed without the intervention of the local host. Reliable communication transports (RD and RC) support memory semantics, while unreliable transports (UD and UC) have limited or no support for it.

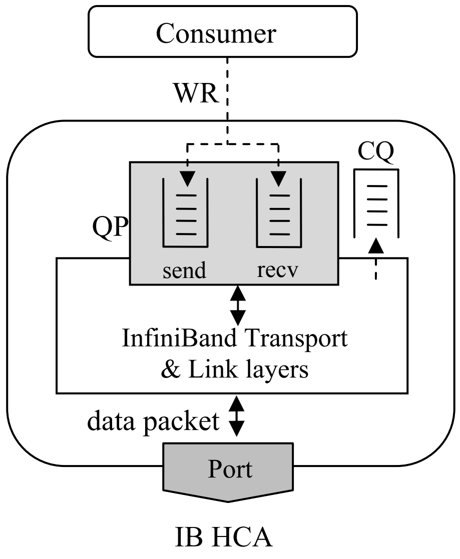


Figure 2.7 InfiniBand HCA communication model

Due to its high bandwidth and low latency, InfiniBand has recently received a significant amount of attention from HPC community, leading to increased usage of this network in the world's top supercomputers [100]. As we will see later, even some Ethernet-based network solutions such as RDMAoE [97] have been proposed that use the InfiniBand processing stack on top of Ethernet link layer for datacenter and HPC applications.

ConnectX [55] is the most recent generation of IB HCAs, released by Mellanox Technologies Inc. [57]. In addition to standard IB features, ConnectX cards support a number of additional functionalities, including a memory-friendly alternative to the IB reliable connection transport called XRC, a reliable multicast capability, and extended quality of service support [55]. The XRC transport service is an extension of the Reliable Connection transport service to improve scalability of InfiniBand clusters [55]. Along with the Shared Receive Queue, this transport service mitigates the number of transport connections required to be established by applications, by specifying/controlling a receiver-side shared receive queue by the senders. ConnectX also allows the combination of XRC with shared send queues, which reduces the required resources even more.

Using the *Virtual Protocol Interconnect* (VPI) [56], dual-port ConnectX cards can also be programmed to operate simultaneously in both 10GE and IB modes. This helps to seamlessly operate a ConnectX card on either or both Ethernet and IB networks at the same time.

2.5.3 iWARP Ethernet

Despite recognized performance inefficiencies, Ethernet currently accounts for more than half of the interconnection networks in the world's top 500 supercomputers [100]. It is due to its easy deployment and low cost of ownership that Ethernet is ubiquitously used in commercial and research clusters, serving HPC and datacenter applications.

TCP/IP is the typical protocol stack used on top of Ethernet. As pointed out earlier, host-based (software) implementation of TCP/IP over Ethernet incurs high communication latency and host

CPU utilization, mostly due to OS-level protocol processing, memory copies and connection-oriented features of TCP protocol. The effect has been exacerbated with the introduction of the most recent generation of Ethernet, 10GE. High throughput demand in 10GE puts even more pressure on the host processor and competes with user applications for available CPU cycles. The potential for starving user applications at high data rates and introducing process skew in synchronous HPC applications has stimulated the use of stateless and stateful offload engines for TCP/IP stacks.

TOEs have emerged as a solution to offload protocol processing from the host CPU to the hardware on the NIC. Essentially, TOEs help decrease the messaging latency and CPU utilization in protocol processing. TOEs reduce the burden on the host CPU by offloading some of the most computationally intensive protocol processing tasks such as checksum calculation, large packet fragmentation, and interrupt processing for reassembly [25]. TOEs decrease the number of memory copies, resulting in lower data transfer time. However, kernel-level memory copies are still required. It is worth mentioning that TOEs have serious critics in OS community due to reasons such as low configurability and manageability, vendor-specific tool requirement, and difficulty for security updates. In addition, the usefulness of TOEs and in general most of the offloaded functionalities is under scrutiny for emerging many-core systems. In such systems, some cores can easily be used to onload the desired functionalities, while making software-level maintenance and management more straightforward. On the other hand, a hardware engine that can efficiently serve all cores on many-core systems is not feasible.

Another important advancement in Ethernet technology is the integration of RDMA technology. The inherent zero-copy feature of RDMA is used in concert with the OS-bypass mechanism introduced earlier. Thus, RDMA effectively helps saving CPU cycles and memory bandwidth, and shortens the communication path. Developing applications with user-level libraries on top of RDMA-enabled NICs allows applications to bypass the OS and communicate directly with the NIC, essentially avoiding the overhead of context switching, system calls, interrupts, and intermediate copies. This also decreases host CPU utilization for network processing. iWARP was the first standardized protocol to integrate RDMA with Ethernet [85].

Proposed by the RDMA Consortium [85] in 2002 to the IETF [38], the iWARP specification defines a multi-level processing stack on top of standard TCP/IP over Ethernet. The stack is designed to decouple the processing of *Upper Layer Protocol* (ULP) data from the OS and reduce the host CPU utilization by avoiding intermediate copies during data transfer (zero copy). To achieve these goals, iWARP needs to be fully offloaded from the application processing core to stateless and stateful TOEs, or onloaded to another core.

As illustrated in Figure 2.8, the top layer of iWARP provides a set of descriptive user level interfaces called *iWARP verbs* that are similar to InfiniBand verbs [31]. The verbs interface bypasses the OS kernel and is defined on top of an RDMA-enabled stack. A NIC that supports the RDMA stack as described in the iWARP standard is called an RDMA-enabled NIC or RNIC. An RNIC implements the iWARP stack and TOE functionality in hardware. Each RNIC manufacturer can provide its own interface and implementation of the iWARP verbs. An example is the NetEffect verbs interface provided with early NetEffect iWARP cards.

The *RDMA protocol* (RDMAP) layer [87] supplies communication primitives for the verbs layer. The data transfer primitives are Send, Receive, RDMA Write, and RDMA Read that are passed as WRs to a QP data structure. The WRs are processed asynchronously by the RNIC, and their completion is notified either by polled CQ entries or by event notification [31].

RDMAP delivers the verbs-layer WRs to the lower layers, in order of their arrival. Send and RDMA Write operations require a single message for data transfer, while RDMA Read needs a request from the consumer (data sink), followed by a response from the supplier (data source) [87]. RDMAP is designed as a stream-based layer. Operations in the same stream are processed in the order of their submission.

The *Direct Data Placement* (DDP) layer is designed to directly transfer data from the application buffer to the NIC without intermediate copies [90]. The packet-based DDP layer matches the data sink at the RDMAP layer with the incoming data segments based on two types of data placements models: tagged and untagged. The tagged model, which is similar to the InfiniBand memory semantics, is used for one-sided RDMA Write and Read operations. It has a sender-based buffer management in which the initiator provides a pre-advertised reference to the data buffer address at the remote side. The untagged model, similar to the IB channel semantics, uses a two-sided Send/Receive model, where the receiver both handles buffer management and specifies the buffer address [90].

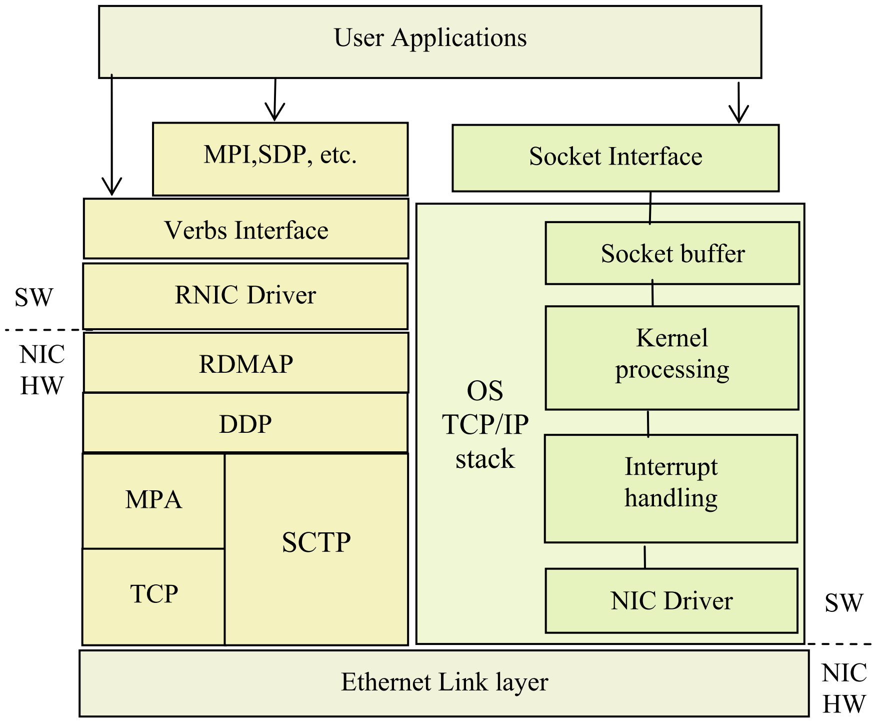


Figure 2.8 iWARP standard stack (left-side) compared to host-based TCP/IP (right-side)

Due to DDP being a message-based protocol, out-of-order placement of message segments is possible. However, DDP assures delivery of a complete message upon arrival of all segments. In the current iWARP specification, DDP assumes that the lower layer provides in-order and correct delivery of messages.

The *Lower Layer Protocol* (LLP) on which the *Marker PDU Alignment* (MPA) [17] layer or DDP layer is running can be either the *Transmission Control Protocol* (TCP) or *Stream Control Transmission Protocol* (SCTP), respectively [69]. Due to the message-oriented nature of DDP, the iWARP protocol requires an adaptation layer to put boundaries on DDP messages transferred over the stream-oriented TCP protocol. The MPA protocol [17] inserts markers into DDP data units prior to passing them to the TCP layer. It also re-assembles marked data units from the TCP stream and removes the markers before passing them to the DDP. The MPA layer is not needed on top of message-oriented transports such as SCTP. For SCTP intermediate devices do not fragment message-based packets as they would with stream-based transports such as TCP. Therefore, the middle-box fragmentation issue that the MPA layer solves is removed.

2.5.4 OpenFabrics and the OFED Software Stack

The OpenFabrics verbs library is the most widely used implementation of InfiniBand and iWARP verbs. Developed by the OpenFabrics Alliance [71], OF verbs (initially known as OpenIB verbs) are included in the *OpenFabrics Enterprise Distribution* (OFED), as a free software stack.

Targeting a wide range of applications from HPC to datacenter and embedded systems, OF verbs are mainly designed as a single open-source RDMA-based transport-independent software stack for Linux and Windows, initially aimed for InfiniBand. However, recently the verbs have been used as an interface to the iWARP standard and RDMAoE as well. Implementing iWARP using OF verbs is considered a significant step in integrating a common messaging layer for different queue-pair-based interconnects. The existence of a single software stack gives freedom to the user to choose the fabric solution. Figure 2.9 shows a general view of the OF software stack. A detailed diagram of the stack can be found in [71].

The *Sockets Direct Protocol* (SDP) [5] shown in Figure 2.9 has been proposed by IBTA for socket-based applications to take advantage of the InfiniBand network offload and RDMA abilities. By default, socket traffic over InfiniBand traverses an emulation of the traditional TCP/IP stack over InfiniBand, called IPoIB. SDP is an alternative, which directly exploits enhanced features of IB such as RDMA and helps to reduce the utilization of the host processor and memory bandwidth in kernel level memory copies and protocol processing of socket based applications. SDP also incurs fewer context switches and interrupts. Although initially proposed for IB, SDP can be used for any other RDMA-enabled network such as iWARP.

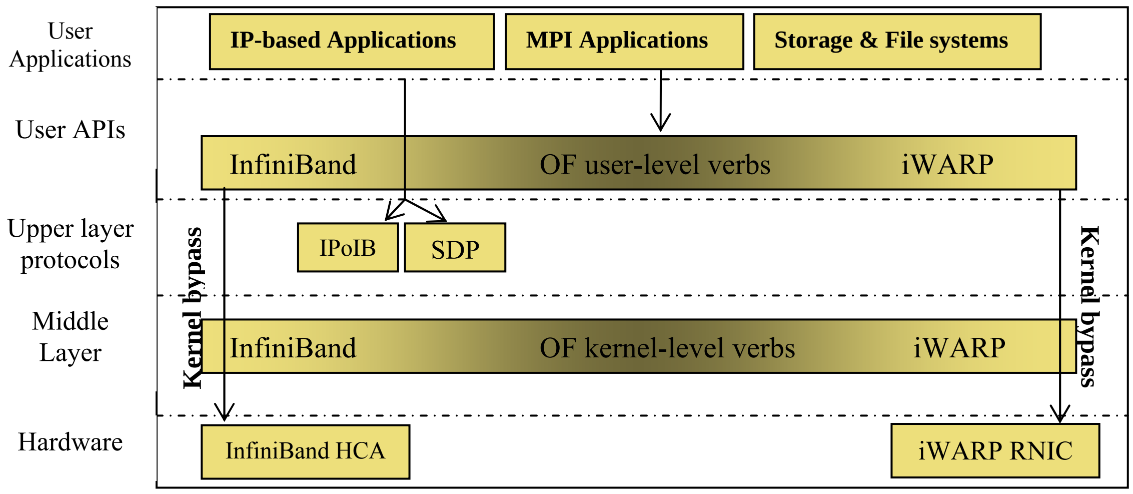


Figure 2.9 A general view of the OF software stack

2.6 Summary

In this chapter I surveyed the communication subsystem in computer clusters. As a part of this review, user and kernel-level message-passing libraries were introduced. The MPI library was presented as the de facto standard in high level message-passing over computer clusters. As mentioned earlier, the main goal of this dissertation is to improve the performance and scalability of computer clusters. As the first step in this direction, the next chapter examines the performance characteristics of modern high-performance interconnects, highlighting their performance benefits and inefficiencies.

Chapter 3

Performance Efficiencies and Shortcomings of Messaging Layer in Modern Cluster Interconnects

As described in Chapter 2, modern interconnects employ innovative techniques such as RDMA and zero-copy, OS bypass using user-level libraries and protocol processing offload/onload to overcome the shortcomings of commodity networks such as Ethernet. In this chapter, I evaluate the communication characteristics of the messaging layer in contemporary cluster interconnects. This evaluation deepens our understanding of the fundamental performance capabilities of these interconnects and identifies their points of strength as well as their performance inefficiencies. It is worth mentioning that this chapter is not concerned with a comparison among the cluster interconnects and their libraries. It rather provides an analysis of their performance characteristics in order to understand their potentials and inefficiencies.

The evaluations are done for both user-level and MPI libraries. At the user level, I benchmark the basic latency and bandwidth for mainstream communication primitives, multiple-connection latency and bandwidth, and cost of memory registration. At the MPI level, I present and analyze the basic latency and bandwidth, multi-pair latency and bandwidth, memory registration and MPI buffer reuse effect, communication/computation overlap, and communication progress ability of modern interconnects [78, 80].

The results confirm that modern high-performance interconnects are providing lower latency and higher bandwidth than the regular Ethernet. Using the same hardware, a 10GE-based MPI is about eight times slower in small-message transfer latency, compared to an InfiniBand-based MPI. The bandwidth is also highly utilized when using modern user-level libraries and offloaded protocol processing. For instance, InfiniBand and iWARP Ethernet utilize more than 90% of their maximum available bandwidth. This evaluation also shows that there exist some points of inefficiency in modern networks, which will be addressed in the other chapters of the dissertation.

The rest of this chapter is organized as follows. Related work is reviewed in Section 3.1. Section 3.2 describes the experimental platform. Basic user-level and MPI performance results are presented in Section 3.3. In Section 3.4 through Section 3.6, I evaluate some in-depth characteristics of the interconnects. Finally, in Section 3.7, I summarize the chapter.

3.1 Related Work

A large body of work exists on the performance analysis of cluster-specific interconnects [7, 10, 34, 50, 51, 76, 103, 103]. In [7] the authors compare Cray 3E, IBM SP, Quadrics, Myrinet 2000 and Gigabit Ethernet. The authors in [51] compare the performance of previous generation of InfiniBand, Myrinet and Quadrics at the user level. In [50] an MPI level comparison of IB, Myrinet, and Quadrics networks is presented. The paper presents the MPI latency and bandwidth, overlap, buffer re-use, and memory consumption. The authors also show the scalability of these networks using some micro-benchmarks.

Another comparison of the InfiniBand network with Quadrics network has been reported in [10]. The authors compare the two networks in terms of memory registration, independent progress, offload and overlap capability. They also present the latency and bandwidth at the MPI level and user level, as well as evaluation of scalability of the library using LAMMPS application [89]. The paper is supplemented with an interesting cost comparison between Quadrics and IB.

A comprehensive performance analysis of Myrinet 2000 is presented in [76]. The paper presents the latency and bandwidth results for the GM user level library as well as MPICH over GM. The authors discuss buffer re-use, overlap, LogP and collective communication results as well as bandwidth for uniform and permutation traffic patterns. The same authors perform a study on two port Myrinet networks with GM2 library in [103]. The latter paper includes some non-data-transfer tests such as memory registration and deregistration costs. It also presents results for GM RDMA performance, MPI level benchmarks for latency and bandwidth, overlap, impact of polling or blocking, and collective communication [103]. The authors also characterize the communication behaviour of some NAS Parallel Benchmarks (NPB) [67] on the Myrinet network.

In [34] performance of MPI over 10GE has been analyzed. In addition to basic MPI micro-benchmark tests such as latency and throughput, the authors report the amount of MPI overhead over TCP/IP library. Their results show that moving towards 10GE does not benefit MPI applications as much as it benefits TCP/IP based applications, primarily because MPI communication semantics is not in match with that of TCP/IP. The authors conclude the significance of revolutionary changes in Ethernet adapters (using TOE or iWARP) to fulfill MPI performance requirements [34].

A few works have been reported on Ammasso RNIC, an early iWARP-enabled Gigabit Ethernet adapter released in 2004 [19, 43, 86]. In [19] the authors reported some basic performance results for this RNIC, comparing MPI latency and bandwidth with those of TCP and iWARP. The paper also presents non-data transfer operation costs such as QP operations and memory registration.

In [43] the authors compare the communication latency over Ammasso iWARP verbs and TCP sockets for wide area network applications. They show that RDMA is beneficial especially under heavy load conditions. In fact, RDMA can provide better communication progress and requires less CPU resources compared to the traditional sockets over TCP/IP. Another work in [86] showed that Ammasso RDMA-enabled Ethernet cards achieve significant performance benefits over conventional Ethernet, with some applications approaching the performance of IB.

A few works also exist on the first generation of 10Gigabit iWARP adapters. In [20] some preliminary performance results are reported comparing the NetEffect's first RNIC [41] on a 133MHz PCI-X bus with a Mellanox 4X IB card on an x8 PCIe bus.

In [66] the authors present the design, implementation and evaluation of the first MPI implementation for iWARP over OF verbs [71]. Their iWARP implementation over Chelsio 10Gigabit iWARP RNIC offers less than 7μs MPI latency and more than 1200MB/s MPI bandwidth on top of RDMA operations. The paper also presents performance for MPI one-sided operations (Put and Get) and some collective operations.

Some research exists analyzing communication/computation overlap and communication progress in cluster interconnection networks [10, 11, 23, 88, 92]. In [11] the authors discussed independent progress, offload and overlap in a conceptual manner, and compared the impact of different MPI implementations on application performance running on two platforms with Quadrics QsNet [75] and CNIC network interface cards (used in ASCI Red machine) [11]. Their results confirm the significant contribution of supporting offload, overlap and independent progress to the performance. Part of my study in this chapter is along the same direction as in [10, 11] with the difference that it analyzes the ability of modern interconnects and their MPI implementations for communication progress and overlap rather than application's ability in leveraging these characteristics. While [50, 103] address combined send and receive overlap, my proposed overlap measurement method in this chapter [82], along with [23, 92] target non-blocking send and receive overlaps separately.

The overlap measurement method proposed in [23] for MPI is a bit different from the method in this chapter in the sense that in [23], at the first step, the communication overhead is calculated, and then the application availability is computed using the overhead value. This is in contrast to the method proposed in this chapter where the overlap and progress ability are calculated by directly measuring the amount of computation that can be overlapped with communication.

Researchers in [92] have presented an instrumentation framework to estimate the lower and upper bounds on the achieved overlap for both two-sided and one-sided communication over IB. In [88] a method is proposed for quantifying the potential overlap of MPI applications. Considering a producer-consumer model of communication, the authors define the potential overlap in an application as the time between the production of a datum at the producer (sender) and its consumption at the consumer (receiver). Experimental results are also presented for some real applications [88].

3.2 Experimental Framework

The experiments for this chapter are conducted using Dell PowerEdge R805 servers. Each machine has two AMD quad-core Opteron 2.0GHz processors, each with a 512KB dedicated L2-cache and an 8MB shared L3 cache on each processor die. Each server has 8GB total physical memory and three x8 PCIe slots. The machines run Linux Fedora Core 5 SMP for x86_64 architecture with kernel 2.6.20.

The iWARP network consists of the NetEffect NE020 10G Ethernet RNICs [42], with PCIe x8 interface and CX-4 board connectivity. OF verbs from OFED v1.4 [71] are used to interface the cards and a Fujitsu XG700-CX4 12-port 10GE switch to interconnect them. I have used single-port Myri-10G NICs with 10Gase-CX4 ports [64] with PCIe x8 interface. Myri-10G NICs operate in MXoE mode using MX-10G version 1.2.8 and are connected to the Fujitsu 10GE switch.

The InfiniBand network consists of *Double Data Rate* (DDR - 20Gb/s per direction) Mellanox ConnectX two-port HCA cards [55] with PCIe x8 interface, connected to a Mellanox 24-port MT47396 Infiniscale-III switch [57]. The OF verbs in OFED 1.4 are used as the user level software interface for ConnectX cards. The ConnectX cards are programmed in the VPI mode [56] in a way that one port could operate in IB while the other port is in 10G Ethernet mode. The

Ethernet tests are run using the 10GE port of the ConnectX cards, but the IB and Ethernet ports are never used at the same time.

MVAPICH 1.1 and MVAPICH2 1.2p1 [68] supplied in OFED distribution are used for IB and iWARP networks, respectively. I use MVAPICH2 because MVAPICH does not support iWARP interface. To have an evaluation of both MVAPICH and MVAPICH2 implementations, I opted to use MVAPICH for InfiniBand. For Myri-10G, I use MPICH-MXoE based on MPICH 1.2.7..1. MPICH2 1.0.8 with Sockets and Shared Memory channel is used for tests over 10GE network.

The Eager/Rendezvous protocol switching point for different networks are shown in Table 3.1. Both MVAPICH and MVAPICH2 implement the Rendezvous protocol using RDMA Write and RDMA Read operations [68]. Unless explicitly specified, the default RDMA Write based Rendezvous protocol is used in these MPI implementations.

Table 3.1 Eager / Rendezvous switching point for different MPI implementations

MPI implementation	Network	Switching point
MVAPICH2 1.2p1	iWARP Ethernet	32KB
MVAPICH 1.1	InfiniBand	9KB
MPICH-MXoE	Myrinet	32KB

3.3 Basic Performance – Latency and Bandwidth

In this section, I investigate the small-message latency and large data transfer bandwidth of the interconnection networks. Section 3.3.1 presents user-level latency and bandwidth results, and Section 3.3.2 presents MPI results.

3.3.1 User level Results

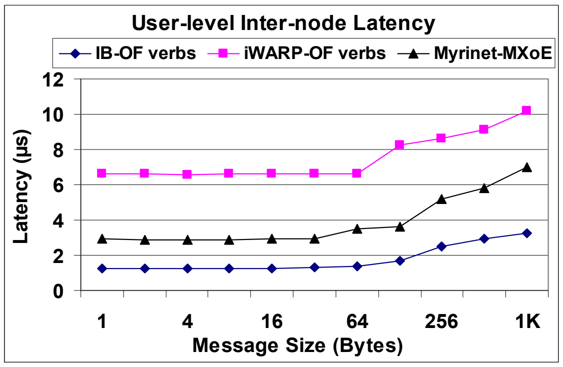
I present the latency and bandwidth of two user-level communication libraries over three interconnects: OF verbs used for InfiniBand and iWARP networks and MXoE for Myrinet cards.

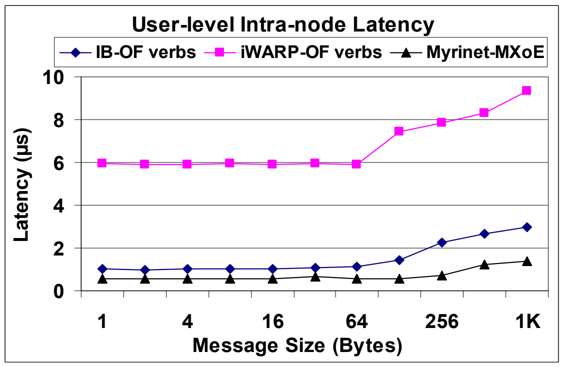
Unlike the MX-10G library, where communication is based on non-blocking Send/Receive operations, tagged (memory semantics) RDMA operations with QP-based communication model form the main communication model in the IB and iWARP protocol stacks. For RDMA operations, the remote memory address tag needs to be exchanged prior to any data transfer. The communication model is one-sided; therefore, an explicit synchronization is required at the end of communication to make sure the data are transferred completely. This kind of synchronization can be done using a Send/Receive (untagged) operation. However, to measure optimistic results in this test, the completion of the RDMA write operations is checked by polling the target buffer.

Figure 3.1 presents the intra-node and inter-node latency of the user-level libraries. For MX-10G, I use the standard *mx_pingpong* latency and *mx_stream* both-way bandwidth tests. For iWARP and IB latency tests, I report the results from standard *ib_rdma_lat* latency and *ib_rdma_bw* both-way bandwidth tests. The user-level small-message latencies for iWARP, InfiniBand and Myrinet are 6.6μs, 1.2μs and 2.9 μs, respectively. The intra-node latencies for IB and iWARP are not very different from the inter-node latencies, because the data travel over the NIC hardware and then loop back into the same node instead of going over the wire. Therefore, it is essentially bypassing the wire and switch latencies. For the Myrinet, however, this is not the case. Unlike OF verbs interface, the MX library uses shared-memory channel for the user-level intra-node communication.

The bandwidth results are presented in Figure 3.2. What we can observe is that while both InfiniBand and iWARP networks are very close to saturation of the maximum available bandwidth (90%) for messages around 4KB and larger, Myrinet network is slowly saturated. The jump in Myrinet's bandwidth for 64KB messages is due to switching to a Rendezvous protocol that directly transfers the data to the receiver, using a direct GET (RDMA Read) operation. The maximum both-way user level bandwidth for iWARP is 2200MB/s, which represents 89% of its maximum available bandwidth (2.5GB/s). The bandwidth of Myrinet gets closer to the available

2.5GB/s by providing more than 2300MB/s bandwidth. On the other hand, the IB verbs saturate 75% of the maximum IB DDR bandwidth, roughly 3100MB/s. The limiting factor here is the PCIe 1.0 maximum throughput for un-encoded data which is 3200MB/s.





(a) Inter-node latency (b) Intra-node latency

Figure 3.1 User-level latency: (a) inter-node and (B) intra-node.

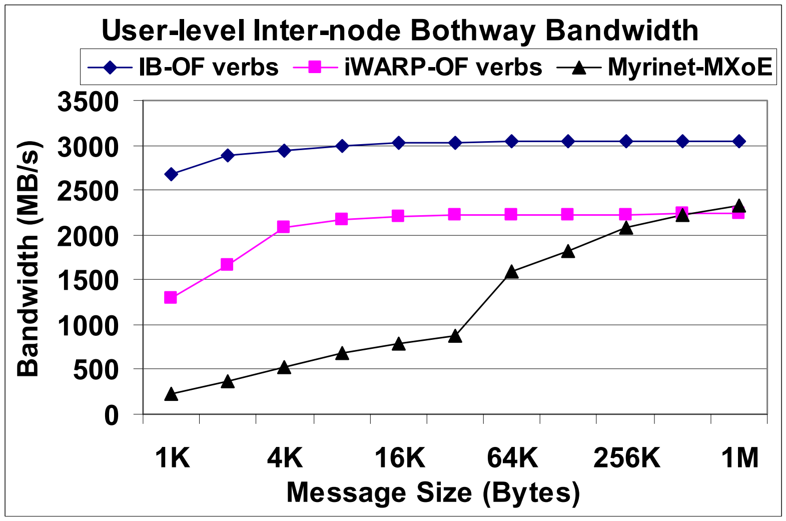


Figure 3.2 User-level bandwidth.

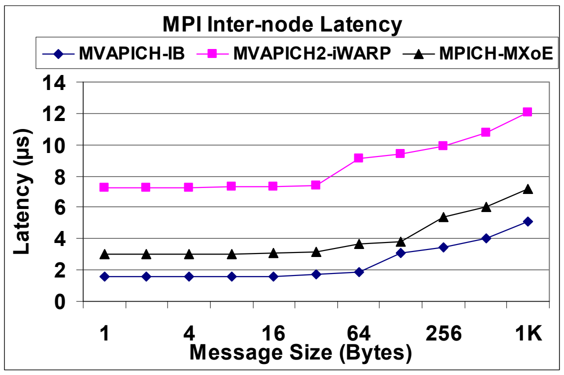
3.3.2 MPI-level Results

In this section, I present the MPI latency and bandwidth results. In the tests, the affinity of processes is bound to cores in order to avoid the overhead of process migration on cache performance.

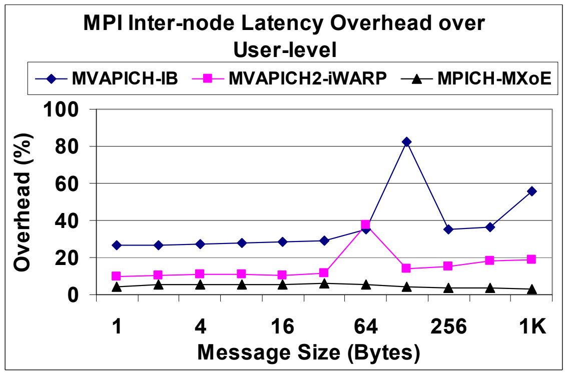
MPI Latency: The latency is measured using a standard ping-pong communication method, as half round-trip time of a message between two processes. Figure 3.3(a) illustrates the inter-node MPI ping-pong latency of the interconnects, for small messages. The small-message latency is around 7μs for iWARP, 1.5μs for MVAPICH-IB, and 3μs for MPICH-MXoE. Compared to Ethernet with the minimum latency of around 13μs over ConnectX 10GE cards (not shown in Figure 3.3), one can observe that there is a significant improvement in small-message latency for modern cluster interconnects, specifically the InfiniBand network.

Figure 3.3(b) shows the latency overhead of the MPI implementations over their respective user level libraries. Since Ethernet does not have a user-level library, the results are not shown for 10GE. Results show that MPICH-MXoE offers the lowest overhead among the interconnects. This is because the MX-10G is the only library with communication semantics close to that of MPI. High overhead for some small messages in MVAPICH-IB shows the high overhead of buffer copies, compared to the small latency of data transfer over the IB network. Part of the overhead is also due to semantics miscorrelation between MPI and the OF verbs user-level library. The reason for peaks in both MVAPICH-IB and MVAPICH2-iWARP overhead curves is due to switching from *Programmed IO* (PIO) based communication to DMA-based communication. In IB, the card uses DMA for 128 bytes or larger messages. This happens at 64 bytes for iWARP. The reason that MPI overhead for these message sizes is high is that for MPI tests the MPI header is added to the user data, making the actual transferred data larger than the PIO limit. Thus, the user-level library is transferring the data using PIO, while the same data bundled with MPI header needs to go over DMA.

Figure 3.4 shows the MPI intra-node latency for small messages. The inter-CMP latencies are measured between processes on different processor chips, while intra-CMP values are for processes on the same processor. Very low inter/intra-CMP latency for small messages confirms that the MPI implementations bypass the network for intra-node communications. Comparing the intra-CMP to inter-CMP communication results, one can confirm that the message latency between two cores on the same processor chip is up to 30% lower than the latency between two separate processor chips. This is because of L3 cache sharing among cores on the same chip. Therefore no memory transfer is performed in cases that the data reside in the cache when both sides are accessing them.

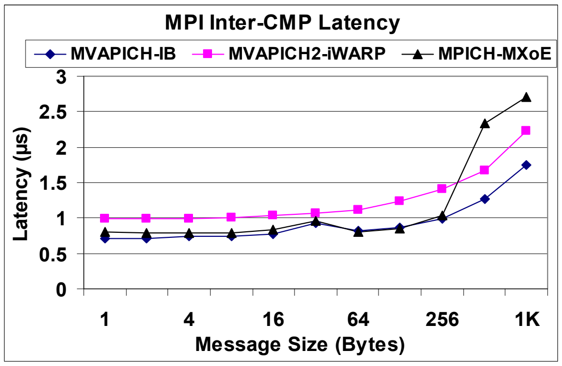


(a) MPI latency

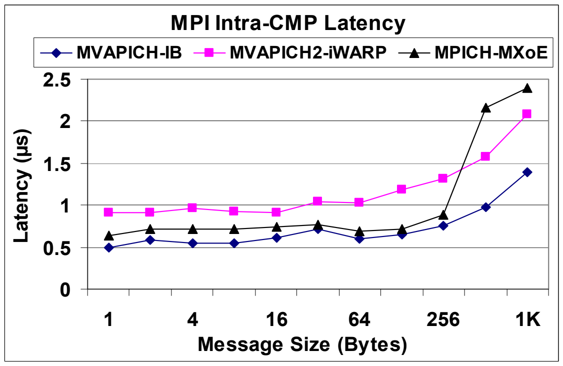


(b) MPI overhead over user-level

Figure 3.3 MPI ping-pong latency and its overhead over user level.



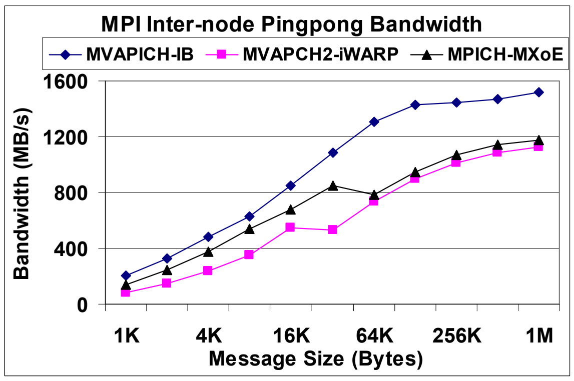
(a) Inter-CMP latency



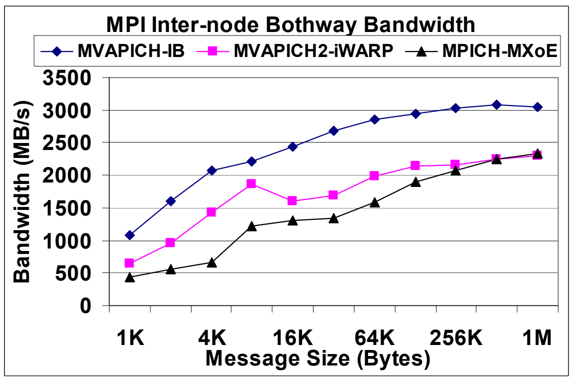
(b) Intra-CMP latency

Figure 3.4 MPI intra-node latency.

MPI Bandwidth: The other characteristic of modern cluster interconnects is their ability to transfer large amounts of data across the network with a high throughput due to avoiding kernel and CPU involvement and using low overhead communication architecture and protocols. In this section, I show how using these interconnects can improve the achieved communication bandwidth. In addition to reporting the ping-pong bandwidth, the MPI inter-node bandwidth is also measured using a standard both-way communication method. The results for ping-pong and both-way bandwidth benchmarks from 1KB to 1MB message sizes are shown in Figure 3.5. In the ping-pong test, up to 1100MB/s for iWARP, 1200MB/s for Myrinet-10G, and 1600MB/s for MVAPICH-IB can be achieved. For all of the three networks, this is very close to (more than 85% of) the maximum one-way bandwidth on our platform.



(a) Ping-pong bandwidth



(b) Both-way bandwidth

Figure 3.5 MPI inter-node bandwidth.

In the both-way test, both the sender and receiver post a window of non-blocking send operations followed by a window of non-blocking receive calls. This puts more pressure on the communication and I/O subsystems. While a maximum bandwidth of 2300MB/s is achievable with the NetEffect iWARP and Myricom's Myri-10G cards, representing 92% network utilization, MVAPICH-IB is nearly saturating the maximum chipset bandwidth of around 3200MB/s. MPI over Ethernet bandwidth results (not shown in Figure 3.5) are notably worse than modern interconnects. Reaching 1400MB/s at 64KB in both-way bandwidth test, 10G Ethernet shows 55% utilization of its available bandwidth. This clearly confirms that the use of advanced communication features in modern interconnects can noticeably increase the communication efficiency of the MPI library.

3.4 Multiple-Connection Scalability

Today, multi-core multiprocessor nodes form the dominant building blocks of high-performance clusters. In such clusters, each core will run at least one process with connections to several other processes. Therefore, it is very important for the NIC hardware and its communication software to provide scalable performance with the increasing number of connections. This is especially important if the NIC is offloading most of the communication processing (which is the case for most modern NICs). Otherwise, the offloaded functionality itself will become a major bottleneck for communication.

Both the iWARP and InfiniBand standards have a QP-based communication model with similar semantics. Both support connection-oriented RDMA operations. Myrinet has an analogous semantics using MX endpoints that work similar to connections in InfiniBand and iWARP standards. With these similarities, it is worth showing how the hardware implementation of these interconnects and their MPI libraries support multi-connection scalability. In this section, the multiple-connection scalability of the networks is evaluated at two levels. I use OpenFabrics and MX-10G libraries at the user level, and MPI non-blocking send and receive operations at the MPI level.

3.4.1 User-level Multi-connection Results

For each network, a number of connections, up to 16 connections, are pre-established between two processes running on two nodes (each node uses only one NIC). Then the communication starts over those connections at the user level. The reason for limiting the number of connections to 16 is the Myrinet's hardware/software limitation for the number of supported endpoints.

I measure the normalized round-trip latency and aggregate throughput over multiple connections. The results show how well communications over multiple connections can be performed in parallel.

To determine the latencies, a ping-pong test is performed using all of the connections in parallel. A fixed number of messages are sent and received over the connections in a round-robin fashion. I vary the number of connections and message sizes and report the cumulative half round trip time divided by the number of connections and message size, named *normalized multiple-connection latency*. For the throughput test, a both-way communication is performed, where each process sends messages to its peer in a round-robin fashion over the established connections. The test lasts for a certain amount of time and at the end, the throughput is reported as the ratio of the amount of data transferred over the communication time.

As shown in Figure 3.6 for small- and medium-size messages, by increasing the number of connections, the normalized multiple-connection latency of small messages for both iWARP and IB cards decreases even for 16 connections. This reflects the ability of the cards to keep up with a number of parallel connections to some extent, for small messages. A relatively fixed latency for messages larger than 2KB on these cards shows a serialization of communications over multiple connections. The results for Myrinet are a bit different. For Myrinet, for all message sizes, the normalized latency decreases well, but only up to four connections (endpoints). This clearly indicates that for all message sizes, Myrinet cards are able to keep up with only up to four connections. Note that the Myrinet results are reported up to 15 active connections, since the MX software on Myrinet cards does not allow more than 15 endpoints.

Figure 3.7 shows the aggregate multiple-connection throughput results for medium to large messages. Obviously, all of the network cards are able to sustain a high level of bandwidth even when multiple connections are actively sending data over the cards. The striking observation is for the Myrinet where the throughput does not scale well with the message size, even with a high number of connections. It only scales slightly up to two connections. For example, with 64KB messages, while InfiniBand and iWARP almost saturate the link over a single connection, Myrinet reaches no more than two-thirds of its link capacity, even with the highest number of active endpoints and using 256KB messages. Comparing this to Figure 3.2, we can see that the throughput for 256KB messages is almost equal to what we observe here for one connection case and this reconfirms that the communication is serialized on multiple connections over this network, especially when using large messages.

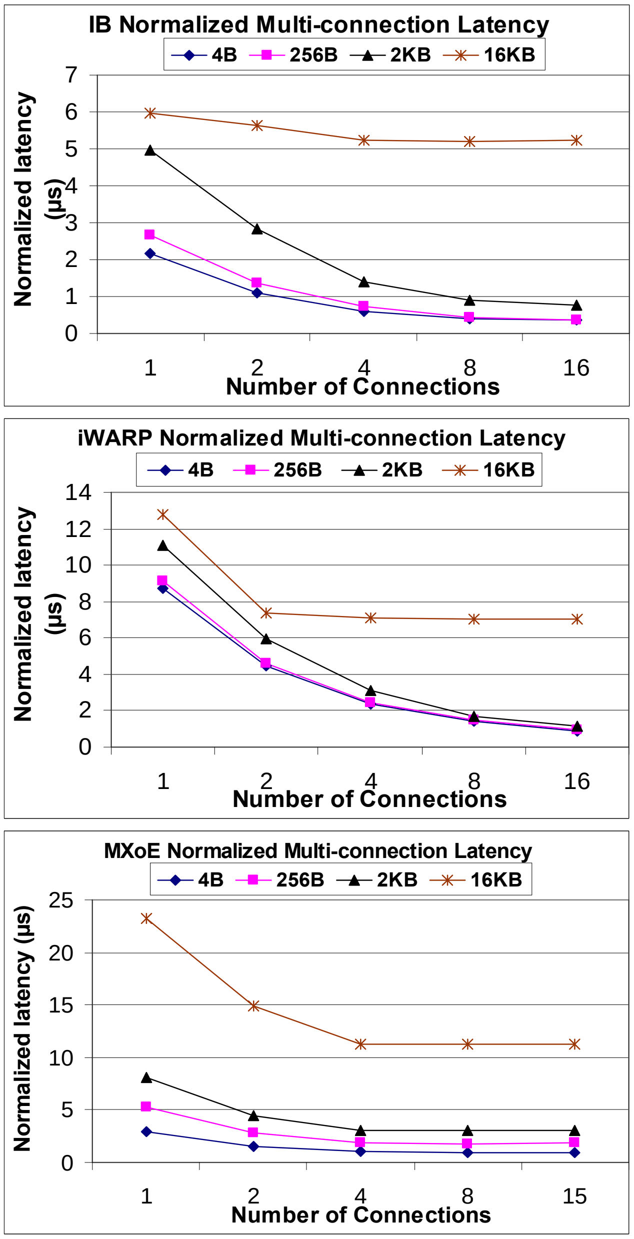


Figure 3.6 User level normalized multiple-connection latency.

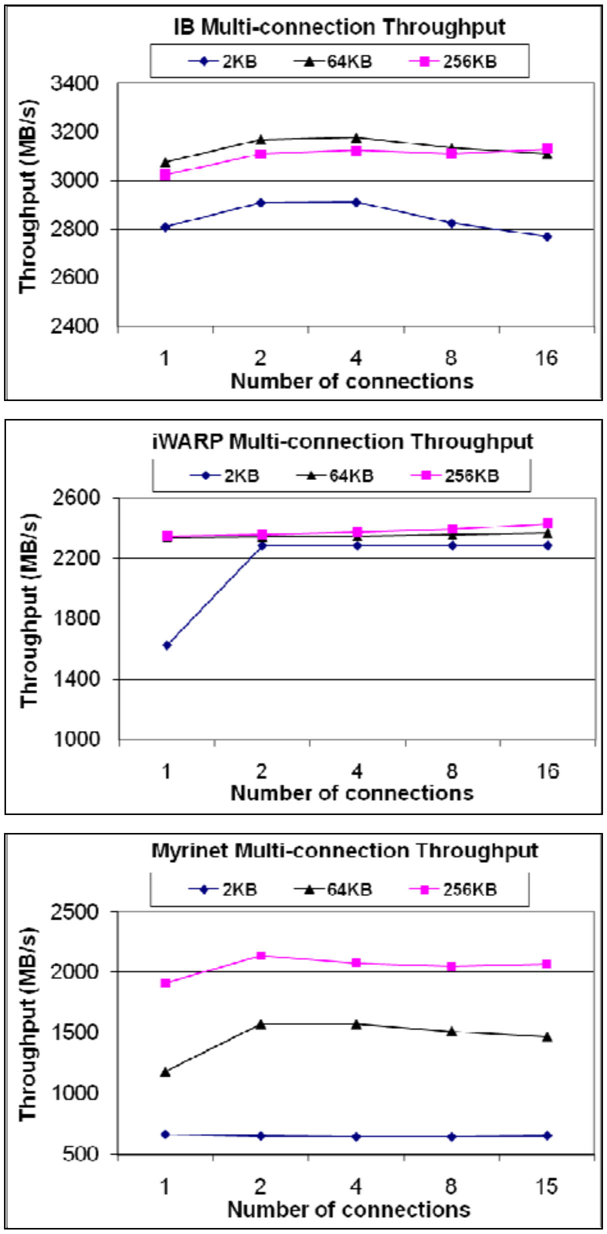


Figure 3.7 User level multiple-connection aggregate throughput.

3.4.2 MPI-level Multi-connection Results

The preceding results were gathered at the user level. In this section, I present the multi-connection scalability results at the MPI level. Similarly, two micro-benchmarks are used, one for measuring the normalized multi-connection latency and the other for the aggregate throughput. Multiple pairs of MPI processes are used, where the two processes in each pair communicate with each other. Due to the connection-oriented implementation of MPI libraries, each pair represents a connection over the NIC. For the latency test, a ping-pong test is performed and the average half-way round-trip time is measured and divided by the number of pairs. For the throughput test, a both-way communication test is performed and the throughout among multiple pairs is accumulated. Since our machines have eight cores per node, I can only experiment with up to eight concurrent communicating pairs. In Figure 3.8, I report the average normalized latency over multiple pairs for all three interconnects. Similar to what observed in Figure 3.6 at the user level, there is a decreasing trend in normalized latency, showing some level of scalability with growing number of active connections on a multi-core platform. Regardless of message size, this scalability almost stops after four pairs for all interconnects.

Figure 3.9 shows the achieved aggregate throughput with different number of communicating pairs. In these tests, each communicating pair performs the both-way bandwidth test. Then, the individual measured bandwidths are added up as the aggregate bandwidth. To ensure all pairs start communication simultaneously, all processes initially participate in a global barrier synchronization. Clearly, when increasing the number of communicating pairs, iWARP and IB bandwidth increases very fast, regardless of the message size. On the other hand, similar to the user-level results, Myrinet shows a slow scale-up trend. Its throughput for a specific message size is almost fixed, regardless of the number of pairs. This implies that the card is carrying data from all pairs into a single communication-processing path. In other words, the communication processing appears to be serialized, not being able to saturate the available bandwidth even with multiple pairs and using relatively large messages. Comparing the user-level and MPI results shows a notable similarity between them. This indicates that the results reflect the characteristics of the network and their user-level libraries in handling multiple connections, even if the connections are handled by a single process pair.

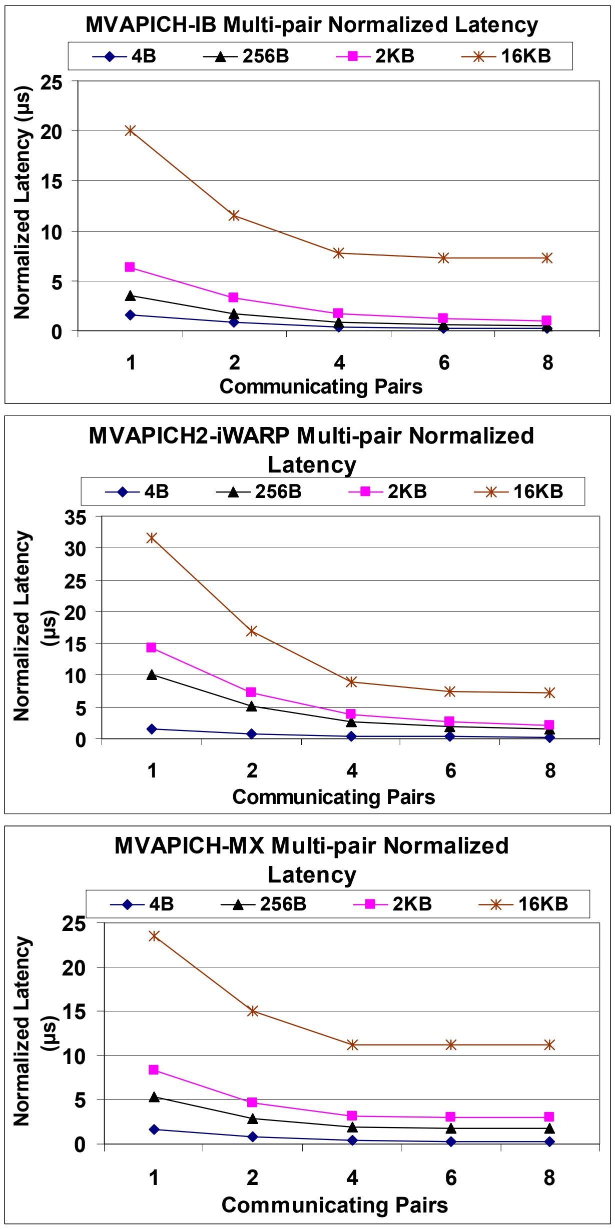


Figure 3.8 MPI-level normalized multi-pair latency.

The pause or slowdown of scaling trend at both MPI and verbs levels in both latency and throughput tests for different networks suggests that the cards' ability to support multiple connections is being saturated for more than a certain number of simultaneous connections. This implies that connection-oriented protocols over modern interconnection networks face serious scalability barriers for future many-core systems with eight or more cores per processor. I will address the connection-based scalability issue for iWARP network in Chapter 6.

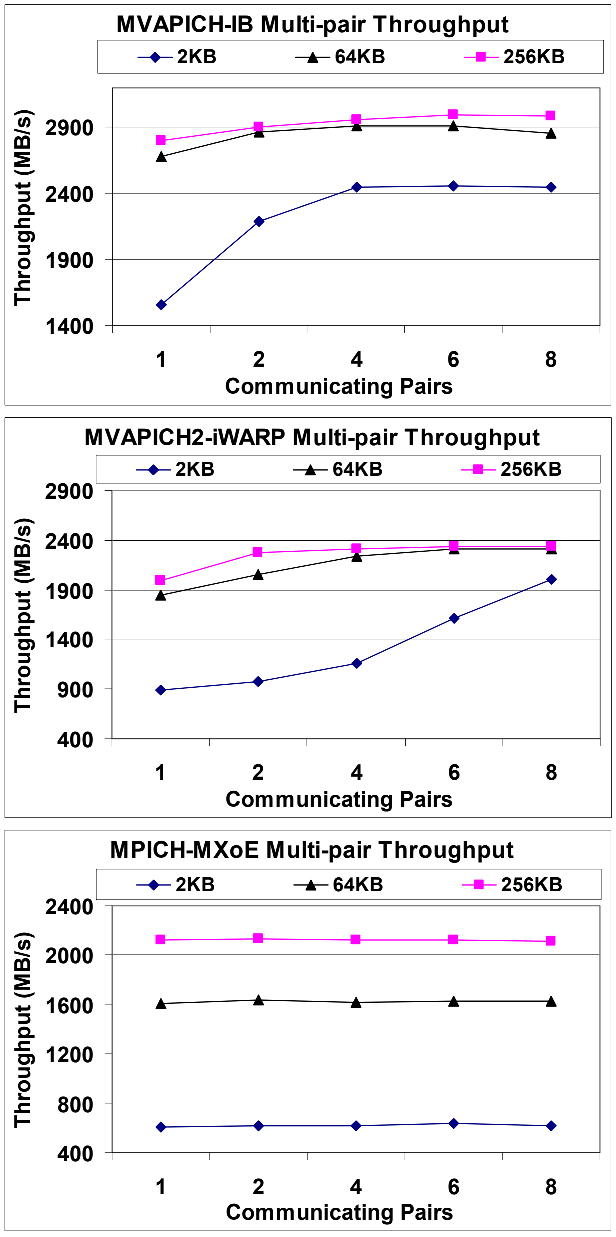


Figure 3.9 MPI-level aggregate multi-pair throughput.

3.5 Effect of Memory Registration and Buffer Reuse

Reusing application buffers for communication is a method that can help avoid a number of buffer management related overheads such as cache and *Translation Look-aside Buffer* (TLB) misses as well as memory registration/deregistration overhead in RDMA-based networks. In this section, I intend to assess the overhead of memory registration in addition to the effect of buffer reuse in RDMA-based interconnects.

3.5.1 Memory Registration Overhead

Memory registration is an expensive process that involves buffer pin-down and virtual-physical address translation [59]. In addition, the registered buffer tag needs to be advertised to the remote node. This is why in the MPI Eager protocol the small messages are copied from the application buffers into pre-registered and pre-advertised intermediate buffers. For the MPI Rendezvous protocol, however, the application buffers are registered to be directly used for zero-copy transfer, since the cost of buffer copy is exceeding the memory registration overhead.

The iWARP and InfiniBand standards require explicit registration of memory buffers prior to using them for communication. However, the Myrinet MX library does not require the user to register/de-register the memory. Instead, the MX library internally performs the registration and de-registration prior to each buffer being used for communication. For this reason, we cannot directly measure the registration/deregistration cost for the Myrinet network. However, an indirect MPI-level measurement can be done by enabling/disabling the registration cache.

Figure 3.10 depicts the memory registration and deregistration costs in IB and iWARP for different buffer sizes. The results clearly show that memory registration is costlier than memory deregistration for iWARP. In fact, using iWARP RNICs, memory deregistration cost is almost independent of the buffer size except for large buffers. However, for InfiniBand, deregistration is somewhat more expensive for small buffers but the trend is similar to iWARP for large buffers, for which registering a buffer is much more expensive than deregistering it.

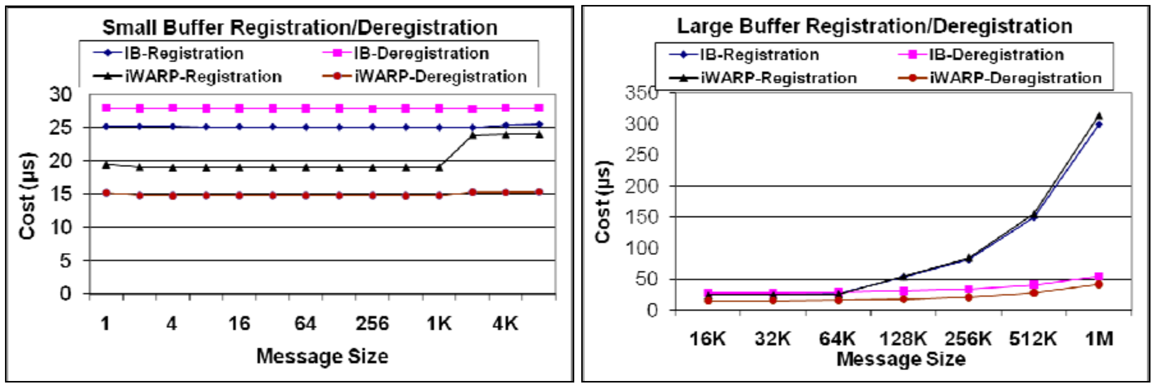


Figure 3.10 Memory registration/deregistration cost.

To avoid re-registration costs at the verbs layer in future buffer reuses, MVAPICH-IB and MVAPICH2-iWARP implementations use a lazy de-registration mechanism [60]. In this mechanism, each previously registered buffer remains registered and its buffer address is stored in a registration cache based on a binary tree data structure. Instead of actually registering and deregistering the buffer, its cache reference count is increased/decreased every time the buffer needs to be registered/deregistered. Only when the registration cache is full, a buffer with zero reference count will be evicted from the cache and then physically deregistered. To keep the information in the registration cache coherent with the OS virtual memory changes, a synchronization with the operating system kernel is required that may negatively affect the performance of the registration cache [102]. The MX-10G library uses a similar internal mechanism at the user level. In case that an application frequently reuses its buffers, the use of registration cache can help avoid repeating the registration of a buffer unnecessarily.

It is therefore worth measuring the actual performance of memory registration cache used in the MPI implementations for these interconnects. This way we can determine how useful the registration cache can be for interconnects that require explicit or implicit memory registrations. Figure 3.11 shows the impact of disabling/enabling the registration cache on the latency and bandwidth of MPI implementations over InfiniBand, iWARP and Myrinet networks, when we reuse all the buffers used for communication in the benchmark. I report the ping-pong results for the latency and the both-way results for the bandwidth. The message sizes used for measurement are all in the Rendezvous protocol range, where buffer registration is performed. As shown in Figure 3.11, the effect of using registration cache on latency and bandwidth is considerable. The effect on Myrinet network, for both latency and bandwidth tests is relatively lower than the other two interconnects, which implies a relatively low buffer registration cost for this network. Since memory registration and the registration cache are internal to the MX library, this feature is a characteristic of the MX library and the Myrinet card.

3.5.2 Effect of Message Buffer Reuse

Reusing a previously referenced buffer has some important performance implications in modern computing and communication architectures. Applications using different message buffers may need to register/deregister their user buffers. Application's buffer reuse pattern affects the registration cache performance, leading to a significant impact on the overall communication performance. Clearly, memory/cache performance, memory address translation overhead and TLB performance are also affected with the application buffer reuse pattern.

In the previous section, I evaluated the effect of memory registration and registration cache on the performance of a micro-benchmark that completely reuses its buffers, and observed that the effect of using the registration cache is significant. In this section, I want to show how reusing communication buffers can affect the performance in the existence of registration cache.

To evaluate the impact of buffer reuse on communication performance, I examine different buffer reuse patterns for the MPI ping-pong test. In this test, a number of separate memory buffers are statically allocated. Depending on the buffer reuse pattern, either a new buffer from the available buffers is selected or the previously used buffer is reused. Figure 3.12 shows the ping-pong latency when changing the buffer reuse pattern from no-reuse (0% buffer reuse) to half-reuse (50% buffer reuse) and full-reuse (100% buffer reuse or always use a single buffer) for each network. Figure 3.12 also shows the ratio of no-reuse latency to full-reuse latency.

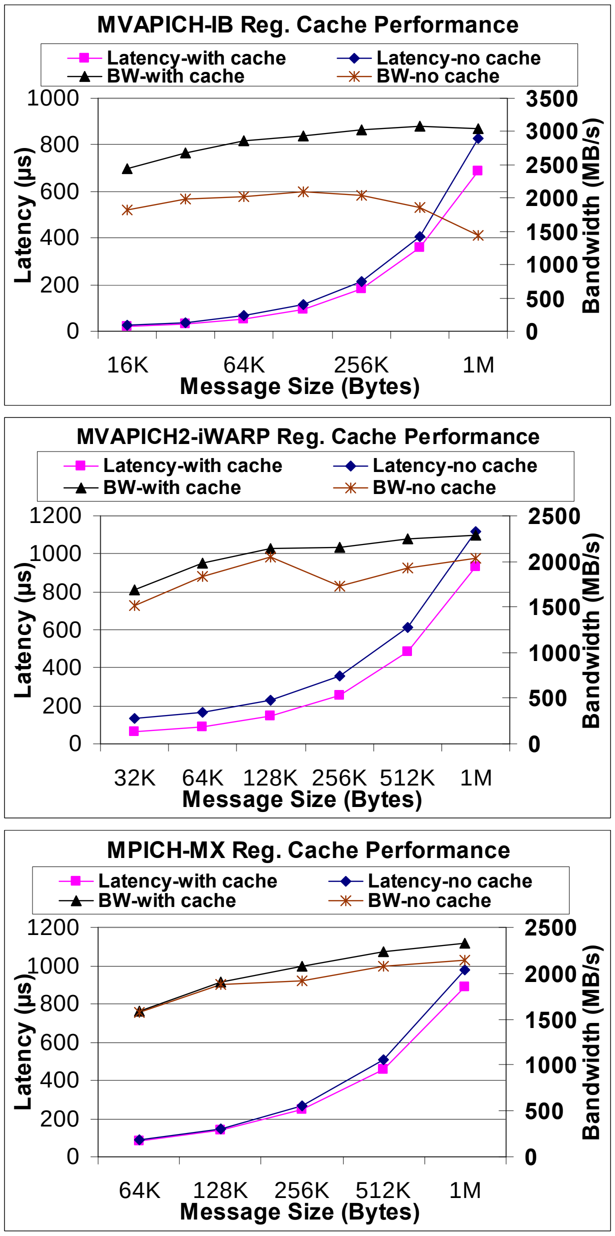


Figure 3.11 Performance effect of Memory registration cache on MPI latency and bandwidth.

For small messages up to 512 bytes, we see less than 20% impact for all the networks. For the Eager size messages, this ratio is less than 1.4 for iWARP, 1.7 for IB and 1.9 for Myrinet. The ratio grows for Rendezvous size messages and reaches 6.7 for IB at 16KB, 2.8 for iWARP at 32KB, and around 2.3 for Myri-10G network at 1MB. This will have a large effect on the Rendezvous protocol's performance, especially when changing the communication buffers more frequently. All of these networks require costly memory registrations for the Rendezvous protocol at the sender and receiver sides when new message buffers are used. However, the impact on the IB is more significant, which can be attributed to its lower base latency compared to the other interconnects.

For large messages, no data copy is performed in the Rendezvous protocol. However, both the sender and receiver processes first search the pin-down cache for a pre-registered buffer. If the buffer is not found in the cache, it needs to be registered. Therefore, with a low buffer reuse profile, a considerable impact due to memory registration cost is observed. We should also theoretically add the possible effect of TLB and cache flushing due to using different user buffers. As observed in Figure 3.12, for all interconnects there is a jump in the ratio of no-reuse latency to full-reuse latency at their Eager/Rendezvous switching point.

On the other hand, for the Eager protocol, source buffers are not directly used for data transfer. Instead, messages are first copied into a previously registered staging buffer. Similarly, messages are first received into pre-registered buffers, and then copied into their destination buffers. This method avoids the registration cost at the expense of two extra copies. So the effect on Eager messages is lower compared to the Rendezvous messages. Nevertheless, there is a slight effect, increasing with the buffer size, which is most likely due to cache and TLB effects.

What we can generally infer from the evaluation in this section is that memory registration is a significant source of overhead on communication latency. On the other hand, buffer copy cost can also be expensive, as it is being incurred on every single message transfer in protocols such as MPI Eager. A careful tradeoff between memory registration and buffer copy can potentially decrease the communication overhead in such protocols. In Chapter 5, an idea will be proposed to avoid one of the two buffer copies in the MPICH Eager protocol, in case that the memory registration cost can be amortized over multiple reuses of a user buffer.

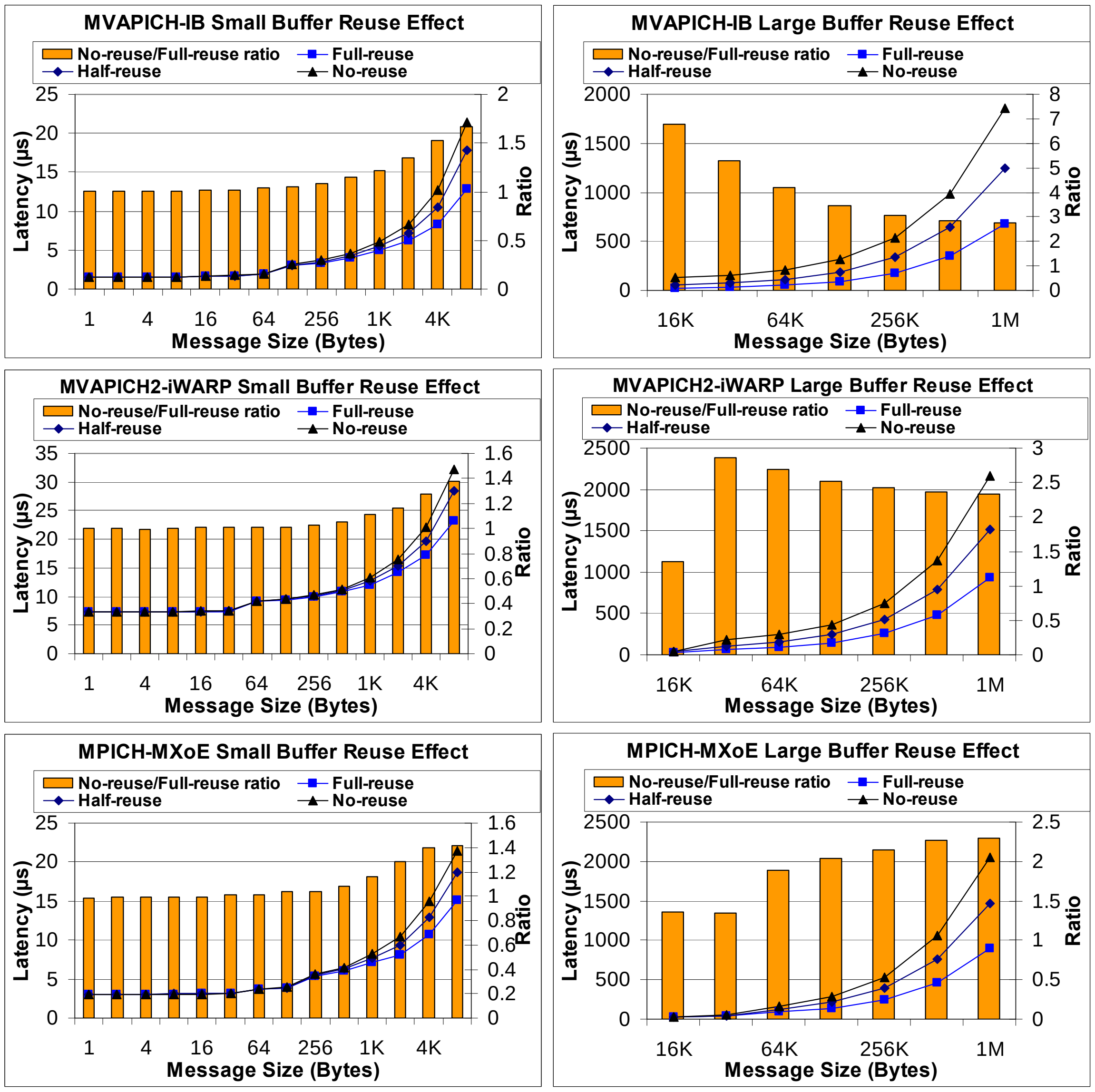


Figure 3.12 Effect of buffer re-use on MPI communication latency.

3.6 Overlap and Communication Progress

Overlapping computation with communication is one of the basic techniques in hiding communication latency, thereby improving application performance. Using non-blocking communication calls at the application level, supporting independent progress for non-blocking operations at the messaging layer, and offloading communication processing to the NIC (or onloading it to the available host core cycles[1]) are the main steps in achieving efficient communication/computation overlap.

Most NICs in modern interconnects are designed to offload most of the network processing tasks from the host CPU, providing excellent opportunity for communication libraries such as MPI to hide the communication latency using non-blocking calls. To utilize the offload engines efficiently, non-blocking communications need to make progress independently. While some MPI implementations support independent progress, others require subsequent library calls in order to make progress in outstanding non-blocking calls. This may have a significant impact on performance when a computation phase follows a non-blocking call. In this section, I intend to assess the ability of modern interconnects to independently make progress in outstanding non-blocking communications and consequently their ability to overlap communication with computation.

3.6.1 Communication/computation Overlap

The overlap measurements are conducted using micro-benchmarks for both MPI non-blocking send and receive operations. I use separate micro-benchmarks to measure send and receive overlap ability. This way we can clearly identify the reasons behind the observed behavior at both sender and receiver sides. In these tests, unlike the method used in [23], I directly measure the amount of computation that can be overlapped with communication. Since the MPI libraries under study use Eager and Rendezvous protocols to transfer small and large messages, respectively, I analyze the results for small and large messages separately.

[1] Although throughout this chapter we rely on offloaded communication processing, the protocol can also apply to onloaded processing, if the onloading is done using extra available cycles on the host cores.

In the send overlap micro-benchmark, the sender uses a loop starting with a non-blocking send call and followed by a computation phase. It then waits for the completion of the message transfer using MPI_Wait(). The receiver blocks on an MPI_Recv(). The time is measured at the sender side. In the receive overlap micro-benchmark, the receiver posts a non-blocking receive call, MPI_Irecv(), and then computes. It then waits for the communication to complete using MPI_Wait(). The sender blocks on an MPI_Send(). Time is measured at the receiver side.

To calculate the overlap in each iteration of the loop, the benchmark measures the portion of the computation phase that can be fully overlapped with communication, without affecting the communication time. Let l_0 be the communication time with no inserted computation, c_m, the computation time inserted in the iteration m, and l_m, the communication time of the iteration m. The amount of computation, c_m, is increased by 10% in each iteration until 10% increase in the original communication time, l_0, is observed (the 10% values are approximate numbers, selected based on empirical observations). The last iteration is the largest m, with $l_m < (1.1)\ l_0$.

Figure 3.13 illustrates the timing measurement of the overlap micro-benchmark. In this model, t_1 is the period that the non-blocking send/receive call is active. In fact, the host CPU is busy with communication processing during t_1. Obviously, the best-case scenario is when the non-blocking call is able to start the communication in the NIC offload engine, before returning from the call. The t_2 period starts when the non-blocking call returns. In this period, the NIC is performing the data transfer using its offload engine, and the CPU is available for computation. Clearly, t_2 is the upper bound for overlap. The t_3 period is the time that the progress engine, called by MPI_Wait(), will complete the communication. It starts right after the computation phase (c_m) or the offload phase (t_2), whichever finishes later.

The original communication latency can be calculated using Equation (3.1). Clearly, if the computation phase in iteration m (c_m) is smaller than t_2, it will be fully overlapped with the communication, and therefore l_m would be equal to l_0, as in Equation (3.2). It is clear that the overlapped computation amount is equal to c_m. On the other hand, if c_m is greater than t_2 (not shown in Figure 3.13), since no more than t_2 time is available for overlap, the start of t_3 period will be delayed, consequently increasing the l_m, as in Equation (3.3). Clearly, in this case, the overlap time is equal to t_2.

$$l_0 = t_1 + t_2 + t_3 \tag{3.1}$$

$$l_m = t_1 + t_2 + t_3 = l_0 \tag{3.2}$$

$$l_m = t_1 + c_m + t_3 \tag{3.3}$$

Using the above equations for iteration m, one can derive the communication/computation overlap time for either of the discussed cases, as c_m - $(l_m$- $l_0)$. Therefore, the overlap ratio is as shown in Equation (3.4):

$$overlap\_ratio = \frac{c_m - (l_m - l_0)}{l_0} \tag{3.4}$$

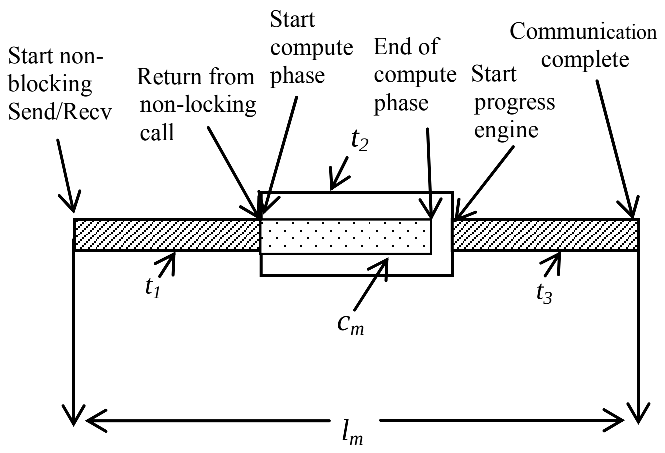


Figure 3.13 Timing model for overlap micro-benchmark.

Send Side Overlap: Figure 3.14 shows the send and receive communication/computation overlap ability of the networks with their MPI implementations. For InfiniBand and iWARP, two Rendezvous protocols are evaluated: RDMA Read based (RGET) and RDMA Write based (RPUT) [98].

One can observe a different behavior for small (Eager range) and large (Rendezvous range) messages. For sending Eager size messages, all networks show a high level of overlap ability. For the Rendezvous protocol, MPICH-MXoE, MVAPICH-IB, and MVAPICH2-iWARP using RGET protocol maintain their high overlap ability, and even reach higher overlap percentages. In the RGET-based Rendezvous protocol over iWARP and IB, the overlap ability increases for Rendezvous-size messages and approaches 100% for 1MB messages. This is because in the RGET Rendezvous protocol, after a one-way handshake from the sender called *Ready-To-Send* (RTS), the receiver uses an RDMA Read to retrieve the message from the sender's memory and therefore the sender is almost free during this operation. Similarly, for the Myrinet network, data are transferred by a direct Get initiated by the receiver using a progression thread, which makes both the sender and receiver processors available [64]. This makes Myrinet send overlap very high for large messages.

On the other hand, both the MVAPICH-IB and MVAPICH2-iWARP results show significant degradation where the send overlap becomes very low for large messages, when using RPUT Rendezvous protocol. This is primarily because RPUT uses a two-way handshake protocol followed by an RDMA Write for data transfer [98]. In this protocol, the sender sends an RTS to the receiver. The receiver will reply with a *Clear-To-Send* (CTS) message that enables the sender to start the data transfer. Therefore, when the sender enters a computation phase after its non-blocking call, the negotiation cycle (reception of CTS) remains incomplete, delaying the start of the data transfer until the next MPI communication call at the sender side.

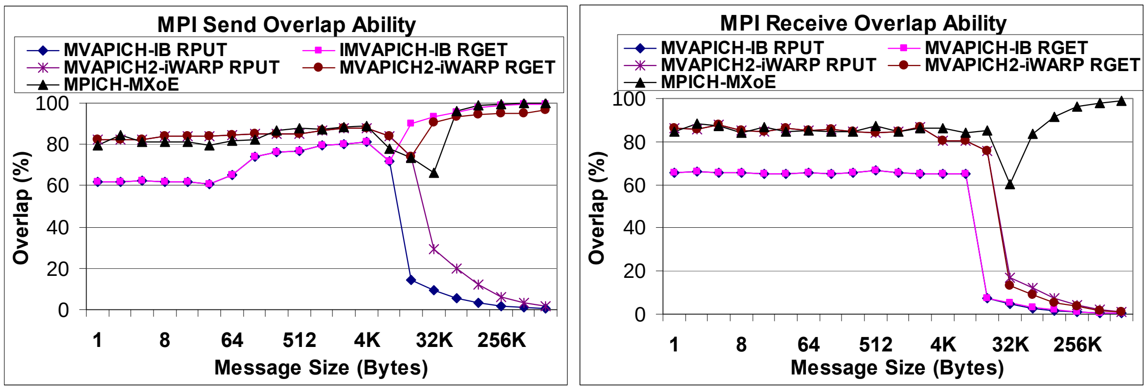


Figure 3.14 Communication/computation overlap.

Receive Side Overlap: In all networks, as shown in Figure 3.14, a high level of overlap ability exists for the MPI receive operation up to the Eager/Rendezvous switching point. For InfiniBand and iWARP, after the switching point, this overlap ability drops to a very low percentage, regardless of the Rendezvous protocol type. Obviously, for large messages, the non-blocking receive calls do not start the data transfer, serializing computation and communication phases. This is due to inefficient use of the MPI progress engine as well as the nature of the used Rendezvous protocol. In case of an early-arrived receiver, no RTS is arrived and thus the communication progress remains after returning from the computation phase. In case that the receiver arrives later than the sender, a progress engine call is needed in the non-blocking receive call to recognize the RDMA-transferred RTS message. Since in none of the Rendezvous protocol implementations such a call exists, the data transfer for two different protocols remains after the computation phase, decreasing the overlap.

A different behavior is observed though for large messages over Myrinet. Since most of the data transfer for Rendezvous range messages is started with a progression thread [64], the receiver CPU is free during the data movement and thus the communication can asynchronously make progress while the CPU is in computation phase.

3.6.2 Communication Progress

Communication progress is called independent if a pending non-blocking communication can make progress without subsequent library calls. This ability helps overlap the on-going communication with computation. The level of independent progress depends on the communication protocols, the progress engine, and the underlying hardware offload or core onloading support. Generally, the progress can be made in protocol negotiations and/or data transfer. Inevitably, with the existence of offload, data movement can proceed independently from the host CPU. However, the start of data transfer and the completion of the communication protocol may still need further library calls.

While most MPI implementations use polling-based progress engines, some have the option of using interrupts for communication progress [46, 98]. In polling-based progress, even in the existence of offload, any computation phase that follows a non-blocking call may delay the progression of the communication protocol that has already been started by the non-blocking call. Although interrupt-based approaches activate the progress engine any time communication progress is needed and enable independent progress while the computation is performed in the CPU, they impose some overhead and fluctuation on the communication time, which cannot be overlooked.

Sender-Side Progress: In the send progress micro-benchmark, the sender invokes a non-blocking send call after synchronization with the receiver. It then spends a certain amount of time in a synthetic delay routine. Unlike the tests for measuring overlap, the inserted computational delay is chosen to be longer than the basic message latency to clearly highlight the progress ability. After the delay, MPI_Wait() is called for the completion of the send request. Then, the sender waits for acknowledgement from the receiver. At the other end and after synchronization, the receiver calls a blocking receive operation and then acknowledges the reception by sending a small message back to the sender. The overall receive latency is recorded for several inserted delay values for each message size. If the latency is increased by the inserted delay, then the communication has not made progress independently.

Figure 3.15 depicts the measured results for the send communication progress in MPI implementations over three interconnects. For InfiniBand and Myrinet, the latency for Eager-size messages (8KB in this test) is not affected by the inserted delay. For iWARP the effect is small, and in fact the message latency is not shifted by the large amount of synthetic delay. Results show that the communication completes during the delay time without any subsequent MPI calls at the sender side.

For the Rendezvous range (256KB in this test), when using RGET protocol, Myrinet, InfiniBand and iWARP show a flat latency curve, meaning that the communication is independently making progress during the delay period. Here, in a one-way Rendezvous negotiation, the receiver retrieves the data from the sender using an RDMA Read upon reception of RTS from the sender [98]. Therefore, the receiver can start the data transfer, independently making progress by utilizing the offloaded protocol engine. That is why we see a flat delay curve for large messages. On the other hand, when using the RPUT protocol for IB and iWARP, the latency is affected by the inserted delay, making the completion of communication happen after the delay. Here, even the completion of the two-step Rendezvous negotiation remains after the delay, thus no data are being transferred during the inserted synthetic delay period.

Comparing the send-side overlap and progress results shows a correlation between the overlap and progress ability of a network in each case. All networks show a relatively high overlap and progress ability when sending small messages. RGET-based Rendezvous protocols show high overlap and progress ability, while RPUT-based Rendezvous protocols over IB and iWARP are relatively weak in communication progress and show low overlap abilities.

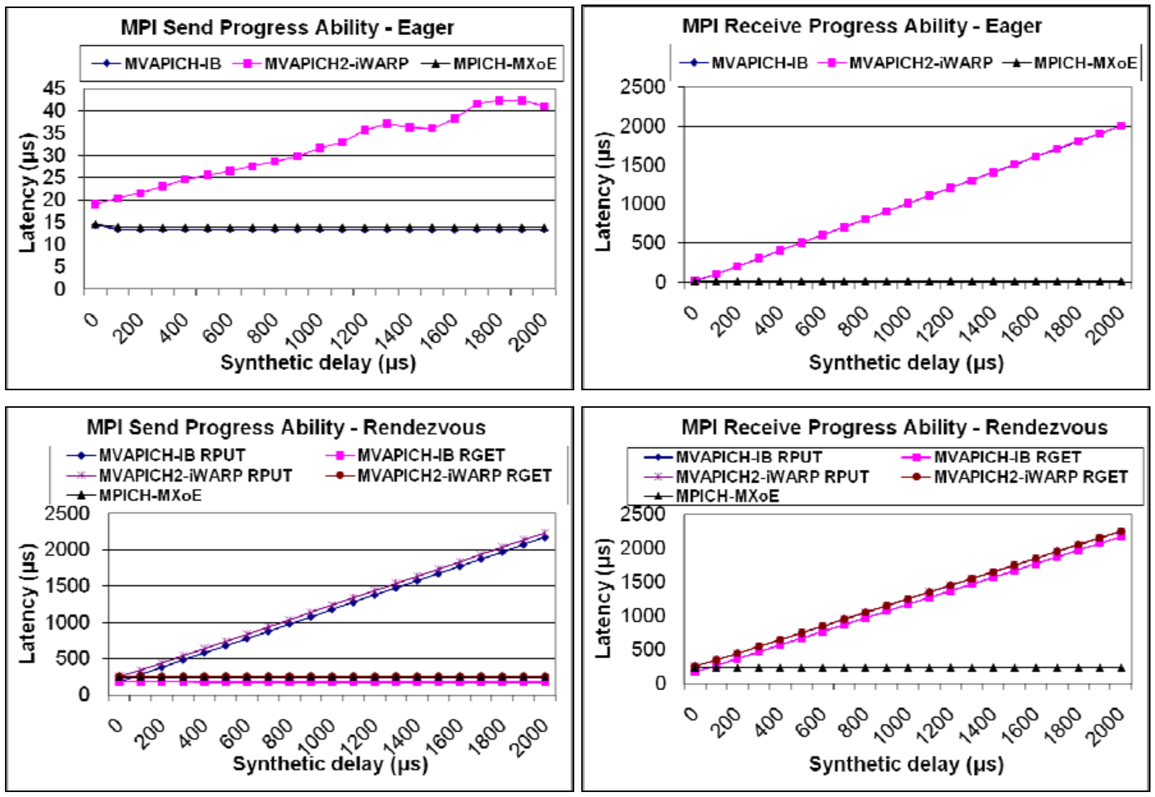


Figure 3.15 MPI communication progress results.

Receiver-Side Progress: A similar test is run for the receive operation. The timing measurement is again performed at the receiver side. After a barrier synchronization, the receiver posts a non-blocking receive call and enters a synthetic delay. During the delay, it also checks for probable completion of the message reception by examining the receive buffer for a certain data value. This helps detect whether the message has been completely received while the receiver is in the delay period. After the delay, MPI_Wait() is called to complete the request. At the send side, a blocking send is called after the synchronization.

The results of the receive progress micro-benchmark for MPI libraries under study are also shown in Figure 3.15. Clearly, except for Myrinet, none of the MPI implementations shows independent progress for receiving Eager-size messages (8KB in this test), and their completion is delayed by the inserted delay time. In general, Eager-size messages in MPICH-MXoE are transferred to a temporary buffer at the receive side upon posting the MPI send call. Similar to what I discussed in Section 3.6.1 about the progress engine call, the observed delay effect is because the reception of the message is not checked in the non-blocking receive call and is left to the progress engine in future MPI calls. Thus, the completion of the receive operation remains for the time that the MPI_Wait() is called, which is in fact after the inserted delay.

Examining the details of receive latencies for Eager-size messages in all networks confirms that only a small portion of the original (non-delayed) communication latency remains after the inserted delay. For instance, the amount of post-delay work for iWARP is only 25% of the original message latency for 8KB messages. This implies that the actual data transfer has been performed during the delay (using direct memory transfer), but the post processing has remained for the MPI progress engine, which is invoked after the delay. This post processing includes finding the arrived message in the unexpected queue, copying it into the user buffer, and finalizing the receive data structure. The reason that we do not see this effect for the Myrinet network is that in case a matching receive is provided when the data arrives at the receiver, the data are directly put into the receive buffer by the MX library. This is unlike RDMA-based transfers in MPI implementations over IB and iWARP for which the arrived data needs to be copied out of a temporary buffer.

The case for the Rendezvous protocol is quite different. For IB and iWARP, the results look similar to that of Eager case, but the latency details imply a different story. In fact, regardless of the Rendezvous protocol in use (RGET or RPUT), all of the message latency remains after the inserted delay. This means that, unlike the discussion above for the send progress, even the data transfer does not start until the inserted delay ends. This is primarily because the non-blocking (send or receive) call does not find the Rendezvous control message (CTS or RTS) to start the data transfer. These messages will be recognized only at the next progress engine execution. The case for Myrinet is different, however. Due to using asynchronous progress using a progression thread, a very small processor overhead on the CPUs at both sides is imposed [64], making the CPUs almost free for computation. The receive side progress results agree with the corresponding receive overlap results explained in Section 3.7.1. Wherever we have high overlap ability, a high level of independent progress exists, at least in data transfer.

In this section we observed that overlap and communication progress can suffer from inefficiencies in the MPI library, despite the fact that the underlying network has offloaded the communication processing to the NIC. In Chapter 4, a method will be proposed to address such inefficiencies.

3.7 Summary

In this chapter, I have investigated the communication characteristics of high-performance interconnects and their messaging libraries. Results show that very low latencies and very high bandwidths are achievable using modern interconnects. Modern networks are also showing different but relatively good scalability in latency and bandwidth when multiple active connections exist over multi-core clusters. However, due to the fast growth in the number of cores per node, such scalability is not adequate for future large-scale clusters with many-core nodes. Chapter 6 addresses the scalability issue specifically for iWARP Ethernet based clusters.

This chapter has also analyzed the various features of MPI implementations over modern HPC interconnects. Due to low network latency, MPI implementations of these modern interconnects are very sensitive to host-based processing overheads such as memory overheads when the buffers are not reused frequently. In addition to low network latency, the high memory registration cost makes the MPI implementations more sensitive to buffer reuse pattern and registration cache performance. As the results confirm, the buffer reuse and memory registration effect is higher for the Rendezvous protocol range (large messages). Obviously, reusing large message buffers in applications plays a significant role in their communication performance on

RDMA-based interconnects. In Chapter 5 I will propose a method to exploit the frequent buffer reuse in MPI applications for small-message transfer protocols.

In this chapter, I also assessed the ability of the modern cluster interconnects and their MPI libraries for communication/computation overlap and communication progress. In terms of overlap, all of the MPI libraries are able to overlap computation with sending small messages that are transferred eagerly. All libraries also show a high level of overlap ability for sending large messages, where a short (one-step) negotiation is used.

The MPI communication progress results also confirm that without independent progress (at least in data transfer) we cannot achieve high level of overlap. MVAPICH2-iWARP and MVAPICH-IB have shown good overlap ability and independent progress for send operations. Myrinet also shows good progress for sending messages, and is able to make independent progress due to its asynchronous progress ability.

All networks except Myrinet show some level of receive progress for small messages, although it is just in data transfer. MPI implementations over both iWARP and InfiniBand show poor receive progress for large messages because the Rendezvous negotiation (reception and recognition of RTS at the receiver side) does not complete in non-blocking calls, preventing the data transfer to start. Myrinet has a better receive progress for all message sizes. To improve the progress and overlap ability of MPI library implementations, a new method is proposed in Chapter 4 that speculatively starts the communication by the receiver, if the receiver arrives earlier than the sender.

In the following chapters, my focus will be on InfiniBand and iWARP networks. The primary reason for not considering the Myrinet network in these works is that the MPI implementation over Myrinet network uses the MX library which has very close semantics to MPI. In fact, looking into the MPICH-MX library, one can see that the design has been optimized for MPI. This leaves limited opportunity to maneuver in the MPICH_MX library for performance optimization. This is unlike the IB and iWARP networks which are designed to efficiently serve different applications, including and HPC and datacenters, and therefore have their distinct communication semantics. The MX library itself is also not open source, and therefore access to the library source is not possible.

Chapter 4

A Speculative and Adaptive Protocol for MPI Large-Message Transfers

As discussed in Chapter 2, MPI implementations typically use different protocols for transferring small and large messages. Examples for MPICH-based implementations are Eager and Rendezvous protocols, respectively. In the Eager protocol (mainly for small messages), the sender sends the entire message to the receiver, where the receiver pre-allocates sufficient buffering space for the incoming message. The Rendezvous protocol is mainly used for large messages, where the cost of buffer copying is prohibitive. The sender and receiver negotiate the availability of the receiver side buffer before the actual data transfer.

As pointed out in Chapter 3, overlapping computation with communication and achieving independent communication progress are two inter-related techniques in hiding communication latency, thereby improving application performance. Modern interconnects with their inherent asynchronous communication ability provide necessary ground for independent progress and overlap. In Chapter 3, it was shown that how the lack of independent progress in the MPI Rendezvous protocol on top of modern interconnects can negatively affect the overlap ability of these networks and their respective messaging libraries. Not supporting such features has an adverse effect on performance, especially for the Rendezvous protocol that involves a negotiation prior to the actual data transfer. This is simply because a non-blocking call may return without completing the negotiation.

While most MPI implementations use polling-based progress engines, some use interrupts for communication progress [94, 98, 101]. Although interrupt-based approaches activate the progress engine any time communication progress is needed, they impose high overhead and fluctuation on the communication time, which cannot be overlooked. In this chapter, I focus on addressing the shortcomings of the current polling-based protocols, by proposing a speculative and adaptive MPI Rendezvous protocol that can increase communication progress and overlap. Unlike the Rendezvous protocols that rely only on the sender to initiate the communication, the proposed protocol in this research lets either the sender or the receiver initiate the negotiation so that the data transfer can be started before the non-blocking send or receive call returns. This will enable overlapping the communication with the computation phase following the non-blocking calls.

The new protocol has been implemented on MPICH2 [4] over 10-Gigabit iWARP Ethernet. The assessment is done using overlap and progress micro-benchmarks for both sender and receiver. Two timing scenarios are considered, where either the sender arrives first in the communication call, or the receiver arrives first. The experimental results indicate that the proposed protocol is able to effectively improve the receiver-side progress and overlap from almost 0% to nearly 100% when the receiver arrives first, at the expense of only 2-14% degradation for the sender-side overlap and progress due to added overhead [83]. The implementation also significantly improves the receiver-side progress and overlap ability when the sender-side arrives earlier, without any negative effect on the sender-side overlap and progress.

I have extended the evaluation of the proposed speculative Rendezvous protocol to some MPI applications in the NPB suite [67] as well as the RADIX application [91] that sorts a series of integer keys using the radix algorithm. The protocol has also been equipped with an adaptation mechanism that disables the speculation when it cannot benefit the application due to the application timing, application communication pattern, and protocol mispredictions. In the experiments, up to 30% reduction in total application wait time has been observed. Meanwhile, the overhead associated with the implementation is shown to be very low, when the adaptation mechanism is in use [81].

The rest of this chapter is organized as follows. Related work is reviewed in Section 4.1. Section 4.2 discusses the proposed Rendezvous protocol. Section 4.3 describes the experimental framework. In Section 4.4, I present and analyze the experimental results. Finally, Section 4.5 summarizes the chapter.

4.1 Related Work

In Section 3.1, I reviewed some of the scholarly work related to overlap and communication progress in MPI. In this section, I will discuss research specifically related to improving the overlap and communication progress in MPI.

On the issue of using interrupts in the Rendezvous protocol, the solution in [1] is based on RDMA Write. They show that the interrupt-based notification tends to yield more overlap ability, compared to the CQ polling method in InfiniBand.

On improving communication progress and overlap, researchers have proposed RDMA Read based Rendezvous protocols for MPI, with the asynchronous progress ability [98]. In their work, a traditional two-way handshake followed by RDMA Write is replaced by a one-way handshake followed by RDMA Read. They also use an interrupt-based scheme with a progression thread to alleviate the bottleneck when the receiver arrives first. Some improvements have been made to the RDMA Read based Rendezvous protocol, aimed at improving its overlap and progress ability and/or decrease its asynchronous overhead [46, 95].

The results in [98] and in Chapter 3 of this dissertation show that shortened Rendezvous protocols using RDMA Read help achieving a good level of overlap and progress at the send side. However, both one-way and two-way Rendezvous protocols are not able to provide independent progress and good overlap for receiving large messages.

The most limitation of the work in [98] is the use of locks (for shared data structures) and interrupts that make the communication time unpredictable and sometimes costly. A more enhanced version of this work has been proposed in [46], where the authors propose a lock-free mechanism in which the auxiliary thread is interrupted and then signals the main thread to take care of the progress when a new control message is arrived. With this method, they are able to achieve higher overlap results compared to [98]. Although they remove some of the interrupts from their earlier work in [98], interrupts to awaken the progression thread are still inevitable due to asynchrony of the progress.

The works in [73] and [95] are perhaps the closest to the research in this chapter. In [73] the author introduces a similar method for improving MPI communication over a cluster of Cell Broadband Engine processors [15]. The cluster under test is interconnected through InfiniBand network and OpenMPI is used for inter-cell communication. Contrary to the work in this chapter that is focused at increasing communication/computation overlap and progress, the work in [73] seeks to improve the MPI communication latency by making the receiver start the communication. The proposed protocol in [73] is basically an amendment to the RDMA-Write based protocol by running the protocol in two steps rather than three steps, when the receiver arrives earlier than the sender. It is not concerned with protocol prediction, mispredictions, or race conditions, because the MPI implementation under study is limited only to a single protocol that is similar to the Rendezvous protocol.

In [95] the authors propose a set of refined Rendezvous protocols for medium and large messages that are designed to provide optimal communication progress and performance under different communication scenarios. Specifically, they propose a hybrid protocol for medium size messages when the sender arrives early, a sender-initiated protocol for large messages when the sender arrives early, and a receiver-initiated protocol when the receiver arrives early. The proposed work uses a counter-based mechanism to take care of race conditions if a protocol decision mismatch between sender and receiver occurs or the control messages cross each other. This is an alternative method of approaching the race conditions for the receiver-initiated

Rendezvous protocol, which requires a counter to be stored and updated for each message envelope in order to match the control messages. When MPI_ANY_SOURCE or MPI_ANY_TAG is used, their receiver-initiated protocol is suppressed and a sender-initiated protocol is used instead.

4.2 The Proposed Speculative Protocol

In this section, I will discuss the details of the proposed MPI Rendezvous protocol. Section 4.2.1 explains the basics of the proposed protocol for two different timing scenarios. I will then discuss the protocol design specification in detail in Section 4.2.2. Section 4.2.3 and Section 4.2.4 cover the methodologies in preventing race and deadlock conditions. Protocol usability is discussed in Section 4.2.5. Finally, Section 4.2.6 describes the adaptation mechanism used to reduce the protocol overhead.

4.2.1 Preliminary Analysis

In the RDMA Read based Rendezvous protocol [98], the sender sends an RTS message to the receiver that includes the sender's data buffer address. Upon receiving the RTS, the receiver process will transfer the data using RDMA Read. Basically, there are two send and receive timing scenarios in the current Rendezvous protocol that could happen at runtime: 1) the sender arrives earlier at the send call; and 2) the receiver arrives earlier at the receive call.

In the first scenario, when the receiver arrives at the receive call, the RTS negotiation message is assumed to be already in the receive buffer. Thus, the data transfer can start immediately. In the second scenario, the receiver calls the non-blocking receive call before receiving the RTS message. It will not see any RTS to start the RDMA Read based data transfer, and therefore starting the data transfer will remain for the progress engine, which is activated either by a costly interrupt [98] or in the next MPI communication call. Consequently, any computation phase after the non-blocking receive call can delay the data transfer. The method that is proposed in this chapter allows the receiver to initiate the communication when it arrives earlier. In the following, I discuss the basics of the proposed protocol for two timing scenarios of the current protocol.

Early Sender: In this timing scenario, the receiver is supposed to transfer the data using an RDMA Read after receiving a matching RTS from the sender. Figure 4.1 depicts the schematic behavior for the current Rendezvous protocol when the sender arrives first. However, due to the one-sided nature of the RDMA operation used to transfer the RTS from the sender, in addition to inefficiency in the original implementation of MPICH2 over RDMA-enabled channels, the receiver is not able to find the RTS which has already arrived. Therefore, a receive request (Rreq) will be enqueued in the MPICH *posted receive queue* (Recvq), leaving the communication progress to a future progress engine call. The results presented for both small and large messages in Chapter 3 (over RDMA-enabled iWARP and IB networks) highlight the existence of such an inefficiency, resulting in non-independent progress for both Eager and Rendezvous protocols. Investigating the issue inside the implementation of MPICH2 revealed that an initial progress engine call is needed to transfer the RTS message from the channel-related buffers into the receiver's Unexq. The receiver is then able to recognize the arrived message and take appropriate action immediately, before returning from the non-blocking call. This initial progress engine call is also beneficial for the send side, as will be described in Section 4.3.2.

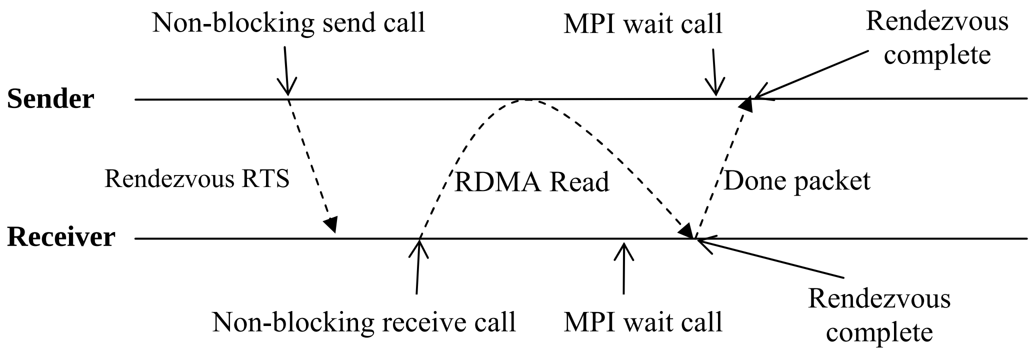


Figure 4.1 Rendezvous protocol in MPICH2-iWARP for early sender.

Early Receiver: In this timing scenario, the receiver enqueues an Rreq in the Recvq and returns from the non-blocking call. The communication will progress when the progress engine is invoked by a subsequent library call. Figure 4.2 depicts the current Rendezvous protocol when the receiver arrives first. In the current protocol, the receiver does not start the Rendezvous negotiation because it is only the sender that knows the communication mode (e.g. synchronous or ready mode) and/or the exact size of the message, which are the factors for choosing the appropriate communication protocol.

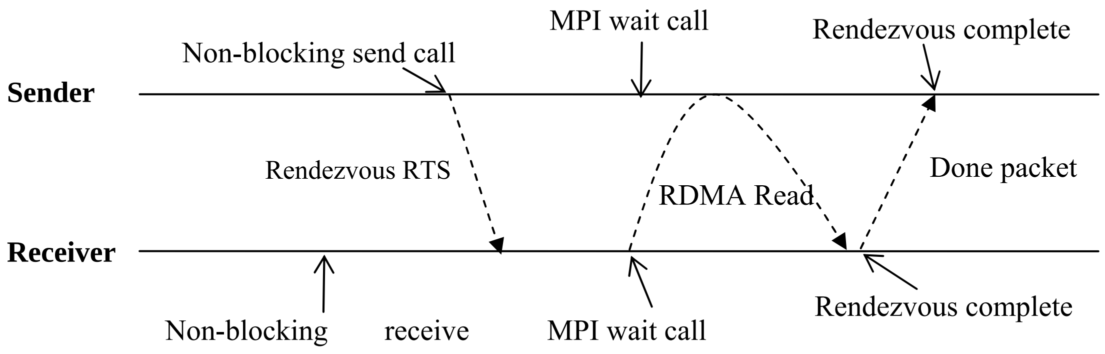


Figure 4.2 Rendezvous protocol in MPICH2-iWARP for early receiver.

In this proposal, the early-arrived receiver predicts the communication protocol based on its own local message size. If the predicted protocol is Rendezvous, a message similar to RTS (I call it *Ready to Receive* or RTR) is prepared and sent to the sender. This message also contains information about the receiver's registered user buffer. At the sender side, if the Rendezvous protocol is chosen, the RTR message (arrived in the Unexq) is used to transfer the data to the receiver using an RDMA Write. Otherwise, if the Eager protocol is chosen, the arrived RTR will be discarded. Figure 4.3 shows a sample timing diagram for the proposed receiver-initiated Rendezvous protocol. Using this protocol, there is more potential for communication progress and overlap of data transfer with the computation phase following a non-blocking send or receive call.

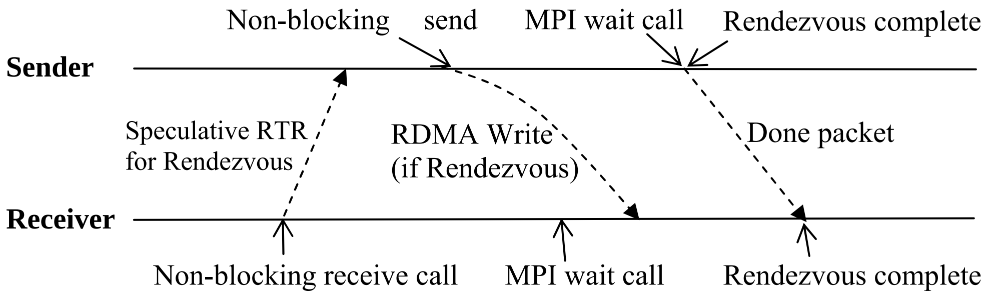


Figure 4.3 Timing diagram of the receiver-initiated Rendezvous protocol for early receiver.

4.2.2 Design Specification of the Proposed Protocol

In this section, I will elaborate on the design and implementation details of the proposed Rendezvous protocol. Basically, the protocol is executed by three MPI library components: non-blocking receive call, non-blocking send call, and progress engine.

Non-blocking Receive Operation: Figure 4.4 depicts a detailed view of the protocol flow when an MPI non-blocking receive is called. Initially, a protocol prediction is performed based on the local message size at the receive side. In the case of Rendezvous protocol, an initial progress engine call is made. Then, for either case of the prediction outcome, the Unexq is checked for a matching message. If an Eager message is found, it will be placed into the user buffer and the communication will be finalized. If a matching RTS is found, the receiver will start the data transfer using RDMA Read. If no matching is found in the Unexq, the receiver side will post the Rreq to the Recvq. If the Rendezvous protocol has been predicted, the local memory will be registered and an RTR message will be prepared and sent to the sender to initialize the Rendezvous protocol, unless producing RTR has been disabled (refer to the adaptation mechanism in Section 4.2.6 and the stop-RTR flag in Section 4.2.3 for details about cases that RTR generation is disabled).

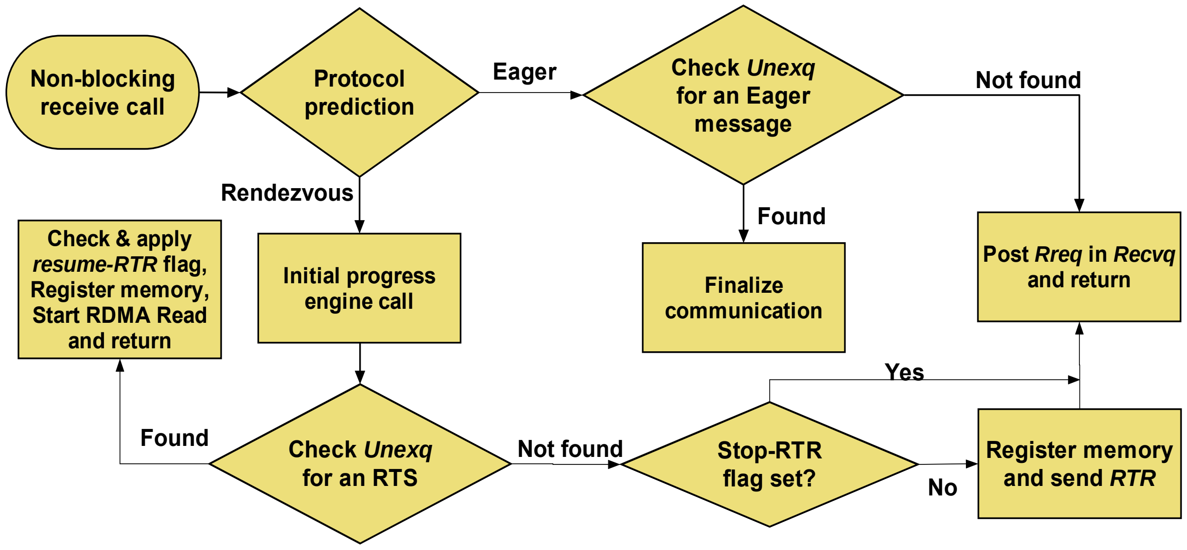

Non-blocking receive call
Protocol prediction
Eager
Check Unexq for an Eager message
Not found
Rendezvous
Found
Check & apply resume-RTR flag, Register memory, Start RDMA Read and return
Initial progress engine call
Finalize communication
Post Rreq in Recvq and return
Yes
Found
Check Unexq for an RTS
Not found
Stop-RTR flag set?
No
Register memory and send RTR

Figure 4.4 Simplified protocol flow for a non-blocking receive operation.

Non-blocking Send Operation: A detailed protocol flow for non-blocking send is shown in Figure 4.5. A non-blocking send call will first decide the appropriate protocol, Eager or Rendezvous. In the case of Eager, the message is transferred eagerly. Since the receiver may mispredict the protocol, it may send an RTR for the current send operation. The mispredicted RTR may mistakenly be used for a future matching send call, posing a race hazard. To prevent such a miss-assignment, the mispredicted RTR should be deleted. A complete description of dealing with race hazards is given in Section 4.2.3. The dashed region in Figure 4.5 which indicates race treatment is expanded in Figure 4.11.

Similar to the case for non-blocking receive, if the Rendezvous protocol is chosen here at the sender side, an initial progress engine call is invoked and then the Unexq is searched for any possibly arrived matching RTR from the receiver. In case a matching RTR is found, the sender will immediately initiate data transfer using RDMA Write, unless a race hazard is detected. The send request (Sreq) will also be enqueued into a send request queue (Sendq) for future reference. The Sendq has been included in the design for posted send calls that have not yet completed their communication. On the other hand, if no RTR is found, the sender will register the local memory and start the negotiation by sending an RTS to the receiver.

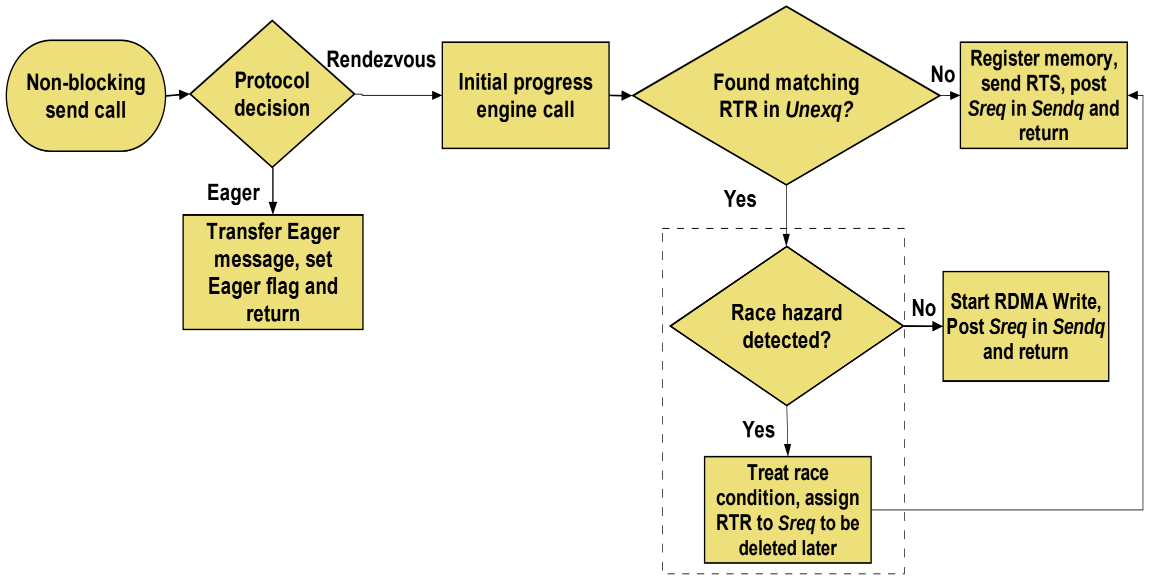

Non-blocking send call
Protocol decision
Rendezvous
Eager
Transfer Eager message, set Eager flag and return
Initial progress engine call
Found matching RTR in Unexq?
No
Register memory, send RTS, post Sreq in Sendq and return
Yes
Race hazard detected?
No
Start RDMA Write, Post Sreq in Sendq and return
Yes
Treat race condition, assign RTR to Sreq to be deleted later

Figure 4.5 Simplified protocol flow for a non-blocking send operation.

MPI Progress Engine: The progress engine in the proposed protocol handles the incoming RTR and RTS messages, and finalizes the communication by actions such as sending *'done'* (final acknowledgement) packets and deregistering the buffer. The RTR processing unit is a new component in the progress engine. The job of the new progress engine is to find a matching send/receive request for the arrived RTR/RTS message in the corresponding send/receive queues, and to start the RDMA-Write/RDMA-Read operations accordingly if a matching request is found.

Figure 4.6 illustrates a detailed behavior of the progress engine according to the proposed protocol. Handling an incoming RTS is as it was before. It is assigned to the first matching Rreq in the Recvq. After removing the matched Rreq from the Recvq, the corresponding data buffer is registered, and the data is transferred using RDMA Read. If no matching request is found in the Recvq, the RTS is placed in the Unexq for future use.

Dealing with RTR is somewhat different, though. If no Sreq is matched against the arriving RTR, the RTR will be placed in the Unexq to be assigned to a future Sreq. Otherwise, the RTR will be assigned to the first matching Sreq in the Sendq that has no RTR assigned to it. The buffer registration and RDMA Write (for data transfer) will take place if no RTS has been sent to the receiver before, and the RTR is not marked for deletion due to race hazard conditions.

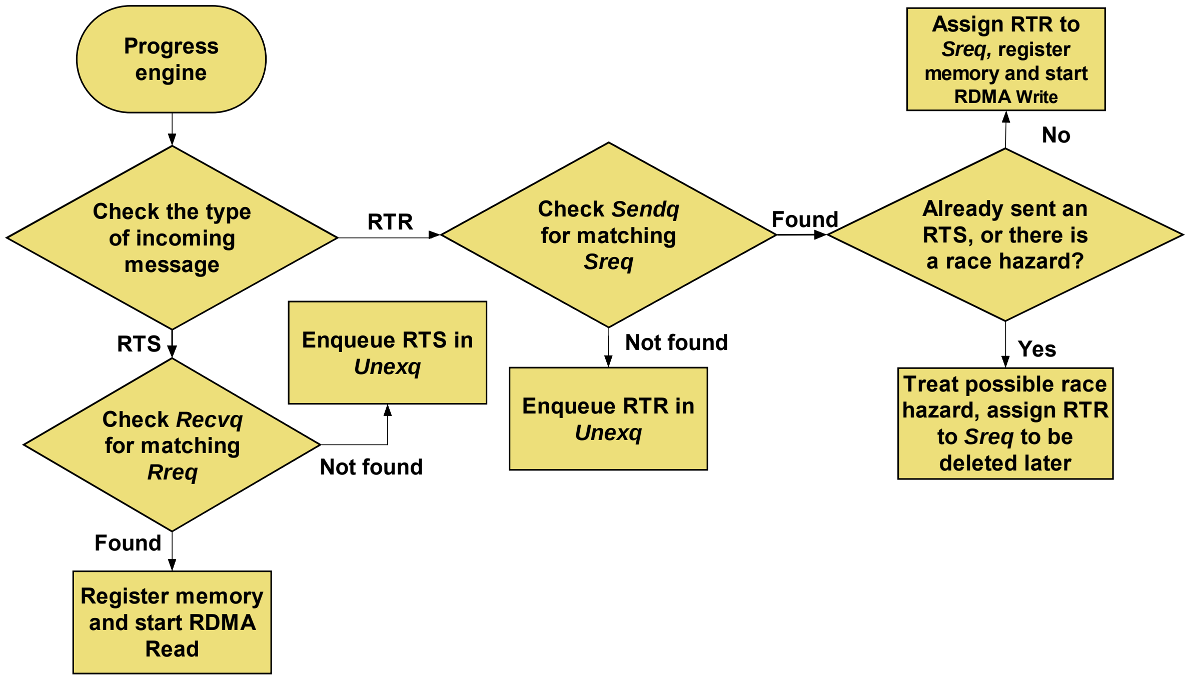

Progress engine
Check the type of incoming message
RTR
RTS
Check *Recvq* for matching *Rreq*
Not found
Enqueue RTS in *Unexq*
Found
Register memory and start RDMA Read
Check *Sendq* for matching *Sreq*
Not found
Enqueue RTR in *Unexq*
Found
Already sent an RTS, or there is a race hazard?
No
Assign RTR to *Sreq*, register memory and start RDMA Write
Yes
Treat possible race hazard, assign RTR to *Sreq* to be deleted later

Figure 4.6 Simplified protocol flow for the progress engine.

4.2.3 Preventing Race Hazards

Unlike the current Rendezvous negotiation model, in which only the sender is responsible for starting the Rendezvous, in the proposed speculative protocol both sender and receiver are able to start the negotiation. Therefore, the new protocol should be checked for race and deadlock conditions. Appropriate modifications and constraints should be considered to avoid undesirable behavior.

Based on the timing of a send/receive call pair, one or both of the calls may start Rendezvous negotiation by sending an RTS/RTR message to the other side. It is also possible that a send/receive request does not produce RTS/RTR due to the reception of RTR/RTS from the other side. Therefore, the proposed protocol with its basic makeup cannot determine the actual source of a received RTS/RTR. Consequently, the actual peer receive/send request cannot be determined. Another reason that makes the protocol undecided and ambiguous about the source/target of a received RTR is the possibility of protocol misprediction at the receiver side. Thus, in either case, the new protocol could cause incorrect assignment of the RTR or RTS messages to the target send/receive requests. In the following, I enumerate different cases that could result in a race condition. I will then propose methods to prevent the hazards for each case.

Ambiguous RTS/RTR Destination Problem: Consider the cases shown in Figure 4.7. All depicted messages have matching message envelopes. In case 1, the first send call finds the arrived RTR and therefore does not send any RTS. As shown, the RTS is coming from the second send call for the second receive call. On the other hand in case 2, since the first send call has not yet received the RTR, it sends the RTS for the first receive call. From the receiver's point of view, both cases are identical. In both cases, an RTR has been sent by the first receive call, and an RTS is received. Therefore, the receiver cannot determine the target Rreq for the arrived RTS message. The same scenarios as shown in Figure 4.7 for RTS messages can generate race hazards for RTR messages as well.

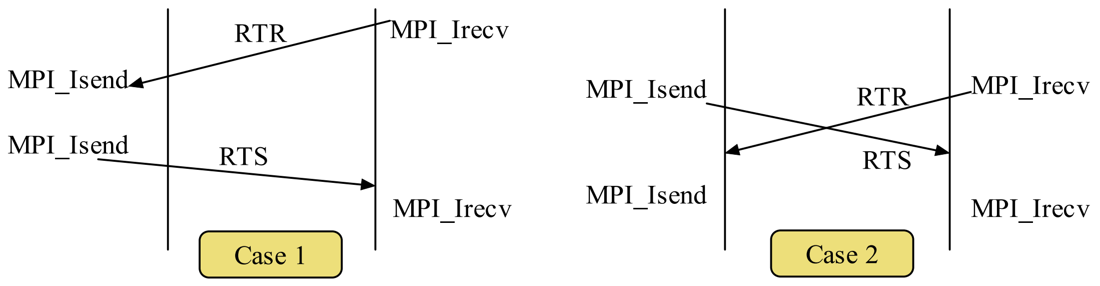


Figure 4.7 An RTS race scenario.

The main reason that the above conditions are hazardous is that the RTS/RTR message assignment policy is not yet defined in the new protocol. In the current Rendezvous protocol, the RTS messages are assigned to the posted Rreqs in the Recvq in order. I keep this policy in the proposed Rendezvous protocol, and extend it to the RTR assignment as well. Therefore, each incoming RTR or RTS/Eager message will be assigned to the first send or receive request in the posted queues, respectively. In addition to this rule, and to avoid the race conditions shown in Figure 4.7, the following rule is amended to the protocol: *a Sreq/Rreq message that finds an RTR/RTS message does not need to send an RTS/RTR message, and should start the data transfer*. However, to prevent the race condition, an acknowledgement (ACK) message will be sent to the other side. With this ACK (to be assigned to its peer Rreq/Sreq), the other side will no longer expect any RTS/RTR from that Sreq/Rreq, and will assign the subsequent incoming RTS/RTRs to the next posted matching Rreq/Sreq. It should be noted that a Sreq/Rreq, which has already sent an RTS/RTR, does not need to send an ACK for an RTR/RTS that it receives later.

Figure 4.8 illustrates how the race conditions in Figure 4.7 can be avoided using the proposed policy. Similar to Figure 4.7, all messages depicted in Figure 4.8 have a matching message envelope. In case 1, the RTR-ACK will be assigned to the first receive call, removing it from the Recvq. Consequently, the RTS will be assigned to the second receive call (as it is now the first matching Rreq in the Recvq). In case 2, the first send call will not send an RTR-ACK, because it has already sent the RTS. The arrived RTS at the receiver side will be assigned to the first receive call (which is the first Rreq in the Recvq). As we can see, the ambiguity surrounding both RTS and RTR assignments are now resolved with the acknowledgement technique.

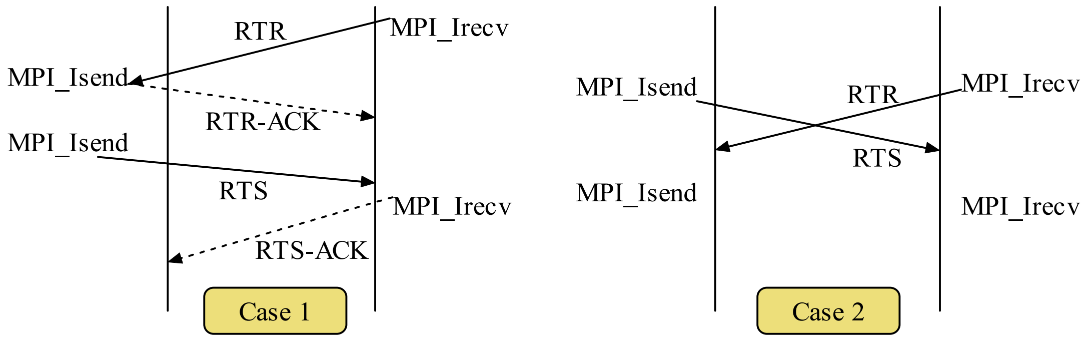


Figure 4.8 Sending acknowledgements to avoid race hazard.

Exceptions to ambiguous destination problem: All send/receive calls can be expected to send a control message (either RTS/RTR or an acknowledgment), except a receive call that predicts the Eager protocol, or a receive call that its message source is indicated as MPI_ANY_SOURCE. If any of these situations happens, an RTR race may occur. This race is related to the case that an Rreq with no RTR can be posted into the Recvq. Consider the scenarios in Figure 4.9. In case 1, the first receive call predicts a Rendezvous protocol and sends an RTR. In the other case, the first receive call is not sending an RTR because one of the aforementioned exceptions happens. In both cases, the target Sreq for the matching RTR arrived at the sender side is not known for the sender. This is because the send side cannot verify if the RTR is coming from the first Rreq (case 1) or the second one (case 2), as it is not aware of the outcome of their prediction.

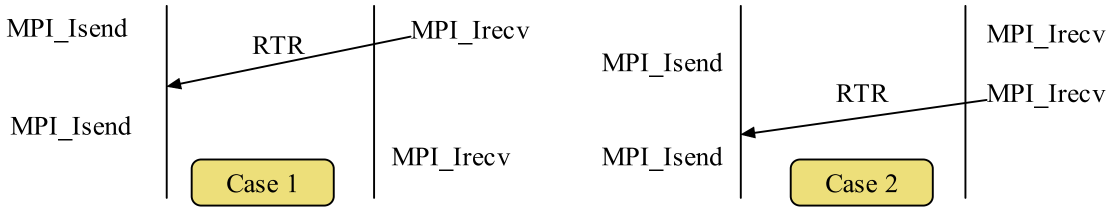


Figure 4.9 Race hazard related to RTR and Eager prediction by receiver.

To cover this race condition, the following policy is proposed: *no matching RTR message will be sent if there is any matching Rreq in the Recvq that has not sent an RTR message*. In such a case, all the incoming matching receive requests have to wait for the send side to start the communication, until the first Rreq (which has not sent RTR) is removed from the queue by receiving RTS or Eager message. Essentially, the race conditions are over by then.

Mispredicted RTR Race Hazard: Mispredicted RTR messages form another source of race hazard, as shown in Figure 4.10. Based on the MPI specification, the size of the receiver buffer can be different (larger) than the actual sender buffer. In addition, the sender may decide to use a protocol for a specific communication mode (e.g. ready or synchronous modes), regardless of the message size. Therefore, the receiver protocol prediction may be different than the sender decision. As shown in case 2 of Figure 4.10, the receiver may mispredict the Rendezvous protocol and send an RTR to the sender, while the sender has decided to use the Eager protocol. On the other hand, in case 1, the receiver predicts the Rendezvous protocol correctly and the RTR is targeted for the second send call. Hence, there is a potential race condition here, because both cases look identical from the sender's perspective, and thus the target Sreq for the RTR is ambiguous at the sender side.

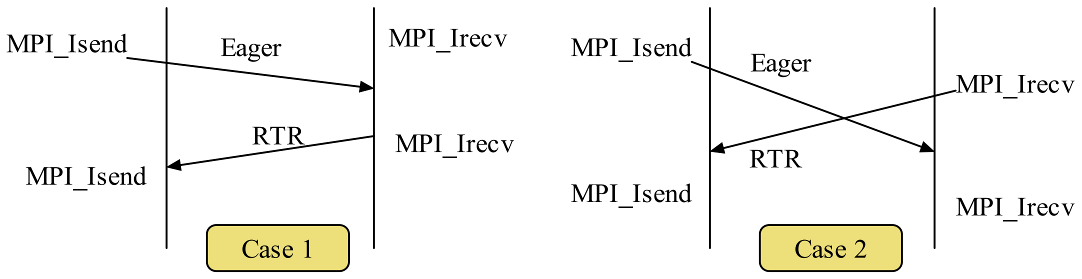


Figure 4.10 A mispredicted RTR race scenario.

To avoid the hazard of mispredicted RTR messages, a flag is introduced, to be called *Eager-flag*. This flag, which is separate for each message envelope, is set by each send call transferring an Eager message. This is a warning for the subsequent matching Rendezvous send calls from the same process to be aware of the possibility of mispredicted RTRs. Since we cannot exactly determine the mispredicted RTRs, the safest method to avoid the race is to drop all RTRs matching with the current Rendezvous Sreq, and start the communication using RTS. Because the current send call cannot make sure whether all previously sent matching RTRs have already arrived or not, the following Eager flag treatment policy is proposed:

- The first Sreq that finds a matching set Eager-flag will start the communication using an RTS and piggyback a *stop-RTR* flag to the receiver side.
- As soon as the receiver side receives the RTS with the stop-RTR flag, it will set a local flag that stops producing RTR for this specific message envelope.
- Once an RTS-ACK or a *'done'* packet (showing the end of the RDMA Read) has been received by the sender, the sender will make sure that all on-the-fly matching RTRs have arrived and no more matching RTRs is forthcoming. Therefore, the sender can now remove all matching RTRs. Once all RTRs are removed by subsequent Rendezvous Sreqs, a *resume-RTR* flag will be piggybacked on one of the matching RTS packets targeted to the other side. This means that all ambiguous RTRs are now safely removed and the receiver side can resume sending RTR messages.

Figure 4.11 shows the completed protocol flow for the send operation. The dashed area in Figure 4.11 shows the expansion of the dashed area in Figure 4.5.

Accumulation of RTR Messages: A less serious issue associated with the mispredicted RTR messages is their accumulation at the sender side. If the sender continues generating Eager messages while the receiver is predicting the wrong protocol, the generated RTRs will be accumulated in the Unexq at the sender side. To prevent this from happening, and also to increase the adaptability of the protocol (in order to decrease the generation of extra RTR messages), the receiver should watch for mispredictions and temporarily disable RTR generation for that specific message envelope.

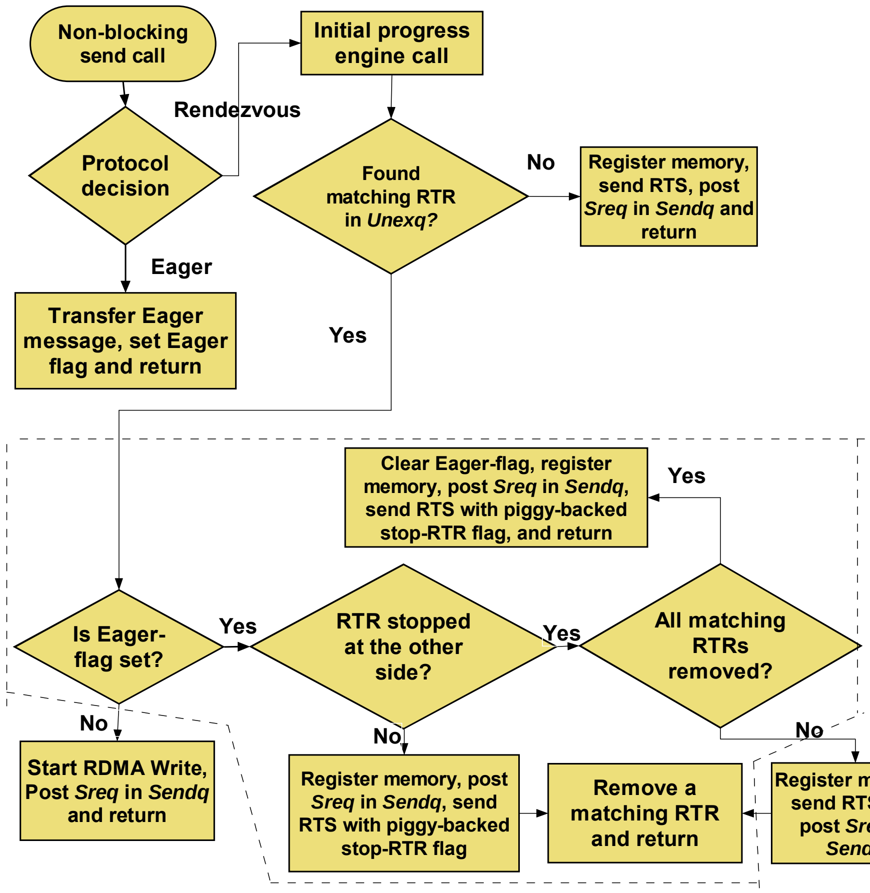

Non-blocking send call
Initial progress engine call
Rendezvous
Protocol decision
Found matching RTR in Unexq?
No
Register memory, send RTS, post Sreq in Sendq and return
Eager
Transfer Eager message, set Eager flag and return
Yes
Clear Eager-flag, register memory, post Sreq in Sendq, send RTS with piggy-backed stop-RTR flag, and return
Yes
Is Eager-flag set?
Yes
RTR stopped at the other side?
Yes
All matching RTRs removed?
No
Start RDMA Write, Post Sreq in Sendq and return
No
Register memory, post Sreq in Sendq, send RTS with piggy-backed stop-RTR flag
Remove a matching RTR and return
No
Register memory, send RTS and post Sreq in Sendq

Figure 4.11 Completed protocol flow for a non-blocking send with Eager-flag management.

4.2.4 Preventing Deadlock Hazards

For a message transfer, deadlock happens when neither the sender nor the receiver proceeds with the communication. In the proposed protocol, there are certain cases that the receiver can post a request into the Recvq without sending an RTR. There are also cases that the sender drops the arrived RTR. However, based on the protocol description in Section 4.3.2 for the send side, either a previously arrived RTR is used or an RTS is sent to the receiver. Therefore, even in the case that an RTR is not sent or is dropped, we will have the current Rendezvous protocol case in place, in which the communication will continue by the receiver using the RTS information and performing an RDMA Read. This leaves no chance for communication deadlock for any messages.

Figure 4.12 shows a coarse view of the Rendezvous protocol flow at the sender and receiver sides. It can be seen that there might be a potential for a deadlock in the receiver-side protocol flow, implying that the receiver can be trapped in a cycle without making progress. However, at the sender side, we do not see such a cycle. The sender will either end up with transferring data using RDMA Write that proceeds with the communication, or with sending an RTS that breaks the receiver-side cycle. Therefore, in either case, the communication will successfully complete.

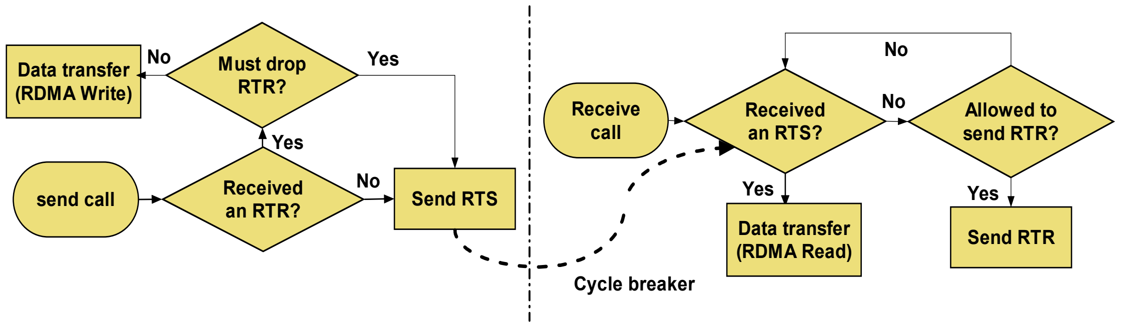


Figure 4.12 Deadlock freedom in send/receive Rendezvous protocol flow.

4.2.5 Usability of the Proposed Protocol

Obviously, only applications that employ non-blocking calls and use the Rendezvous protocol for communication will benefit from the improvements achieved by using this protocol. In addition, the benefit is mostly in cases where the non-blocking receive calls are posted earlier than their peer send calls. Since the new protocol tries to initiate the communication using the early non-blocking calls rather than the MPI wait calls, appropriately interleaving communication calls (i.e. non-blocking send and receive and MPI wait calls) with computation phases will maximize the gain by increasing the communication/computation overlap. On the other hand, there are some cases that an application may not benefit from the proposed protocol and may even suffer from its overhead:

- If the application timing and synchronization is such that the receive calls are rarely invoked before their peer send calls, the proposed protocol will not help the application's communication progress much. In such a case, the application performance may even suffer from the protocol's speculation overheads (e.g. race and deadlock prevention, acknowledgement messages, etc.).
- In some applications, due to very tight synchronizations, it is possible that the peer send and receive calls arrive almost at the same time, and therefore both RTR and RTS messages are generated. These messages will then cross each other, and based on the protocol design the RTR will be dropped. While an application with such a timing does not need the RTRs, the overhead of RTR handling is inevitable.
- When an application does not overlap its computation phases with communications, any effort to improve MPI library for communication progress and overlap is useless, and in fact it may adversely affect the application performance due to implementation overheads. It should be noted though that the discussion of the utilization of overlap in modern scientific applications is beyond the scope of this research.

4.2.6 Protocol Adaptation to Avoid Speculation Overhead

To reduce the amount of overhead imposed on applications that do not benefit from the new protocol, an adaptation mechanism is embedded in the proposed protocol to stop the speculation and revert the protocol to the current RDMA Read based protocol. Basically, there are two adaptation techniques: *static* and *dynamic*. In the static adaptation, the execution of an application is profiled, and based on the profiling outcome it can be determined whether the new protocol is useful or not. The speculation can be permanently disabled for such an application if the protocol is not useful. In dynamic adaptation methods, the behavior of the speculative protocol is monitored during the application runtime, and the speculation is dynamically disabled when it is not useful.

I propose a method for dynamic adaptation. I opt for a metric that can reflect the behavior of the speculative protocol and its likely impact on application performance. The new protocol generates an RTR message when a Rendezvous protocol is predicted. However, when the prediction is incorrect, or when an RTR message is crossing a peer RTS message, the transferred RTR is discarded and consequently the system resources are wasted for its generation and transmission. Therefore, I use the *RTR usage factor* (the percentage of the received RTR messages that have been used by the sender) in this study, as it is a metric that shows how accurate and useful the speculation is. The adaptation mechanism compares the RTR usage factor for each message envelope with a threshold value, and stops the speculation if the RTR usage is lower than the threshold. Here is a complete description of the algorithm:

- Each process monitors the RTR generation and usage statistics. A search table (a hash table for large systems and a linear queue for small systems) is used to store the RTR statistics for each message envelope.
- Before preparing and sending an RTS, the sender will check the RTR usage factor. If it is lower than a certain threshold, the speculation is stopped. In addition, as no RTR will be generated the send-side initial progress engine call will also be stopped. A flag will be piggy-backed on the outgoing RTS to inform the receiver about the decision, and so the RTR generation for that message envelope will be stopped at the receiver.

4.3 Experimental Framework

The experiments were conducted on four Dell PowerEdge 2850 SMP servers, each with two 2.8GHz Intel Xeon processors (with 1MB L2 cache) and 2GB of DDR-2 SDRAM. The machines run Linux Fedora Core 4 SMP with kernel version 2.6.11. The iWARP network consists of NetEffect NE020 10GE RNICs, each with a PCIe x8 interface and CX-4 board connectivity. A Fujitsu XG700-CX4 12-port 10GE switch connects the nodes together. I use MPICH2-iWARP, based on MPICH2 1.0.3 over NetEffect verbs 1.4.3. By default, MPICH2-iWARP uses the Rendezvous protocol for messages larger than 128KB.

4.3.1 MPI Applications

I use four applications in evaluating the proposed Eager protocol: CG, BT and LU benchmarks from NPB 2.4 benchmarks [67], and RADIX application [91]. CG solves an unstructured sparse linear system using the conjugate gradient method. CG mostly uses MPI send/receive and barrier operations [24]. LU is a simplified compressible Navier-Stokes equation solver. LU mostly relies on MPI blocking (and a few non-blocking) send/receive, and some broadcast, all-reduce and barrier operations [24]. BT is a block tri-diagonal solver that mostly uses non-blocking MPI point-to-point calls [67]. RADIX is an integer sorting application [91] with different MPI point-to-point and collective calls.

4.4 Experimental Results and Analysis

I have implemented the proposed Rendezvous protocol on MPICH2 over NetEffect 10-Gigabit iWARP Ethernet. Micro-benchmarks as well as application benchmarks are used to evaluate the proposed protocol's ability for overlap and independent progress in comparison to the current MPI Rendezvous protocol. Section 4.4.1 discusses the micro-benchmark results. Section 4.4.2 presents and analyzes the application results, and Section 4.4.3 covers the overhead of the proposed protocol.

4.4.1 Micro-benchmark Results

In this section, I use micro-benchmarks to evaluate how the new protocol may affect overlap and communication progress. The micro-benchmarks are similar to the progress and overlap benchmarks described in Section 3.6. However, I have revised these micro-benchmarks so that we can run them in two timing scenarios: when the sender is always forced to arrive earlier (to use RTS), and when the receiver is forced to arrive earlier (to use RTR). Assessment is done at both sender and receiver sides.

Receiver Side Overlap and Progress: Enhancing the receiver side Rendezvous overlap and progress ability has been the main goal of this work. Figure 4.13 and Figure 4.14 compare the overlap and progress ability of the proposed Rendezvous protocol with the current protocol for the two timing scenarios. The performance results show that the new protocol has been highly successful in its objectives. Note that Figure 4.14 presents the results only for 1MB messages. Similar results have been observed for other long messages.

Improving receiver-side overlap ability from less than 10% to more than 80% (more than 90% for most message sizes) in both timing scenarios can be clearly observed in Figure 4.13. In the case that the sender arrives first, the initial progress engine call at the receiver side has helped to find the already arrived RTS message. Thus, we have now achieved the expected level of receiver side overlap and progress ability. The overlap is more than 80% and the progress benchmark latencies are not affected by the inserted delay, confirming a complete and independent progress.

The main achievement of the proposed Rendezvous protocol is for the other scenario, in which the receiver arrives earlier. Since the early-arrived receiver speculatively sends an RTR message, the late-arrived sender will find the RTR and start the communication (unlike the current protocol where the early-arrival receiver would start communication when it receives the RTS from the sender after its computation phase). Therefore, complete progress and almost full (more than 92%) overlap is achieved during the computation phase.

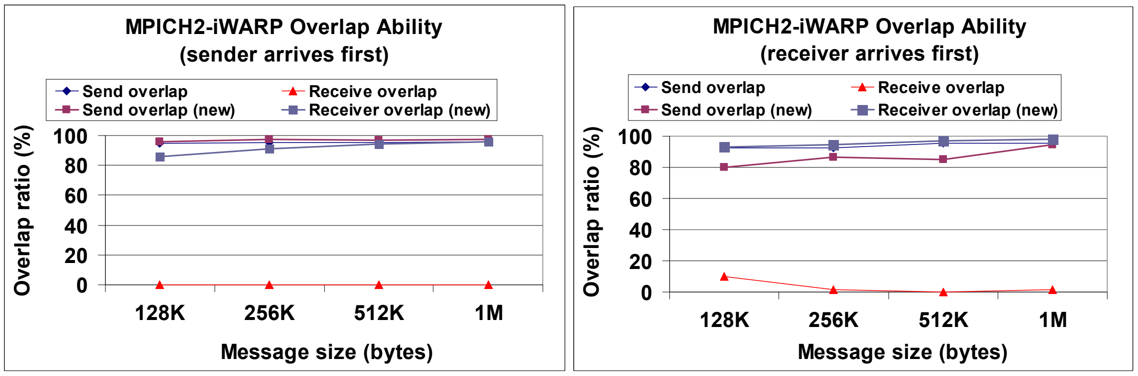


Figure 4.13 Current and new Rendezvous overlap ability for two timing scenarios.

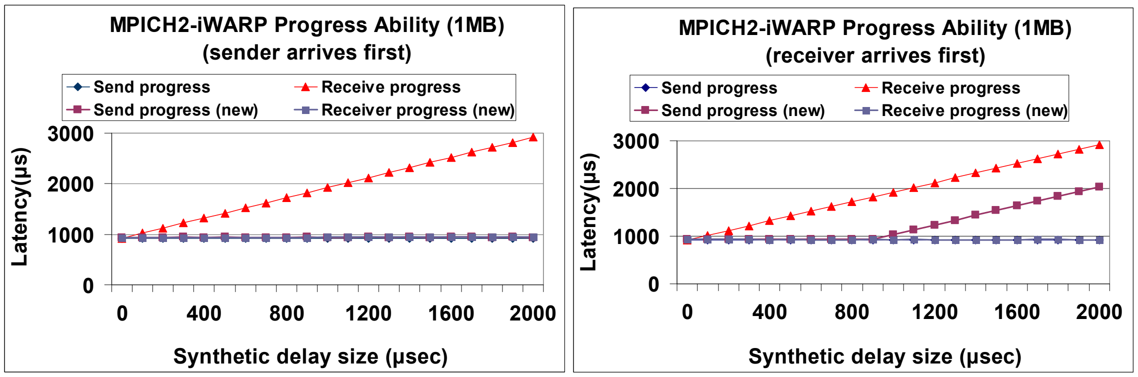


Figure 4.14 Current and new Rendezvous progress ability for two timing scenarios.

Sender-side Overlap and Progress: Based on the protocol details, as we have targeted the receiver side overlap and progress, we do not expect a major change in the sender-side overlap or progress ability. The results in Figure 4.13 and Figure 4.14 for the send overlap and progress comply with our expectations.

Comparing the current and the new send overlap results, we observe that except for some message sizes when the receiver arrives first, the overlap is almost at the same level or even better. This is a good achievement given that the protocol has added some extra overhead at the send side. For the case that the receiver arrives earlier, the sender-side overlap has dropped slightly (2-14% based on the message size). This can be due to the required RTR processing in the non-blocking send operation before launching the RDMA Write, as well as some post processing after the delay for sending the *done* packet.

The sender-side progress results comply with the overlap observations. When the sender arrives first, the same negotiation scenario occurs as in the current protocol. Therefore, we see similar progress results. However, just like the overlap results for the case that the receiver arrives earlier, the new sender-side progress is a bit worse. Although the corresponding results presented in Figure 4.14 suggest that the latency has been shifted by the inserted delay (after a certain point), examining the details of latency results suggests that only 4-15% of the whole communication latency (depending on the message size) remains after the inserted delay. These results confirm that MPI has been able to make progress in more than 85% of the communication, which is in harmony with the corresponding overlap results in Figure 4.13. The reason behind this can be attributed to the fact that the latency is measured at the receiver side to make sure the message has completely arrived. When the receiver arrives earlier and sends an RTR, the sender will start the RDMA Write after finding the RTR. Thus, the RDMA Write will be overlapped with the sender-side synthetic delay (inserted after the non-blocking call). However, the receiver does not finish the receive operation until receiving a done packet from the sender, which is sent after the delay. This affects the latency in the cases that the inserted delay is more than the message latency, which can clearly be observed in Figure 4.14.

Exchange Model Micro-benchmarks: In this section, I evaluate the effect of the proposed protocol on a data exchange computation model that is used in many scientific applications. In each iteration of this model, two (or more) processes produce their data, and then exchange the data with the designated neighbors (one or two neighbors in our implementation), and finally consume the exchanged data. The pseudo-code for the exchange model micro-benchmarks is shown in Figure 4.15. In this code, the computation phases are interleaved with the non-blocking communication calls in order to maximize the communication/computation overlap at the MPI code level. Two different code alternatives are used to examine the effect of the new protocol when the order of send and receive calls is changed. In the pseudo-codes, *Pack (data$_i$)* represents the pre-processing of data usually performed in some applications to pack non-contiguous data buffers to be sent together to the receiver.

Exchange model 1	**Exchange model 2**
Timer_start	Timer_start
Produce(data$_1$)	Produce(data$_1$)
Loop i=1 to iterations	Loop i=1 to iterations
MPI_Irecv(data$_i$)	Pack(data$_i$)
Pack(data$_i$)	MPI_Isend(data$_i$)
MPI_Isend(data$_i$)	If(i>1) Consume(data$_{i-1}$)
Produce(data$_{i+1}$)	MPI_Irecv(data$_i$)
Wait(i,Isend)	Produce(data$_{i+1}$)
Wait(i,Irecv)	Wait(i,Isend)
Consume(data$_i$)	Wait(i,Irecv)
End Loop	End Loop
Timer_stop	Consume(data$_{iterations}$)
	Timer_stop

Figure 4.15 Pseudo-code for data-exchange benchmark.

I have measured the iteration time when using the current Rendezvous protocol and compared it to the new protocol with and without using the adaptation mechanism. The results are shown in Figure 4.16, for a number of communication/computation ratios. Communication time is calculated as the iteration time when no computation is performed. For the overall computation time, I add up the times associated with the produce and consume operations. For the sake of simplicity, produce and consume times are considered identical in these tests.

As shown in Figure 4.16, for the benchmark Model 1 that the non-blocking receive is called before the non-blocking send, due to benchmark timings most of the generated RTRs are used because the RTRs are recognized by the sender before it generates any RTS. This makes the late sender transfer data using RDMA Write, while the produce operation is in progress, which helps improve the overlap and reduce the whole iteration time. With small communication time, the overlap opportunity is low and so is the improvement. As the ratio of communication time to computation time increases, the amount of overlap also increases, leading to up to 32% improvement in the benchmark time. This is because when the communication/computation ratio is increased, a larger ratio of communication (compared to the computation time) is overlapped. On the other hand, with very high communication, the whole computation can be overlapped. However, because most of the iteration time is spent in communication, the latency hiding does not cause significant improvement in the overall benchmark time. Clearly, since in micro-benchmark Model 1 the RTR usage is high, the adaptation method does not take effect and the results for adaptive and non-adaptive cases are very close.

In micro-benchmark Model 2 where the send call is before the receive call, in all cases, the RTS messages are recognized in non-blocking receive calls, because of using early progress engine call inside the non-blocking receive operations. Therefore, data transfer is started before the produce phase and the communication is overlapped with computation. However, with different communication times, different improvement ratios are obtained. With similar reasons discussed for the exchange Model 1, the improvement is low when communication/computation ratio is either very high or very low. The maximum improvement is about 20% when the communication time is 50% of the computation time.

Although in both cases a fixed 128KB message size is used, in micro-benchmark Model 2 the maximum overlap occurs in a lower communication/computation ratio (50% compared to 80% in benchmark Model 1). The reason is that in Model 2, more computation is overlapped with the communication phase due to the fact that the data transfer from both sides is performed using RDMA Read (initiated using RTS messages). On the other hand, in most of Model 1 iterations, RTRs are used and therefore data is transferred using RDMA Write. Thus, with the same message size, in Model 2 a longer communication phase (RDMA Read compared to RDMA Write) is available. Moreover, because I use the entire pre-measured communication time to calculate the communication/computation ratio, the maximum improvement occurs in a smaller communication/computation ratio, compared to Model 1.

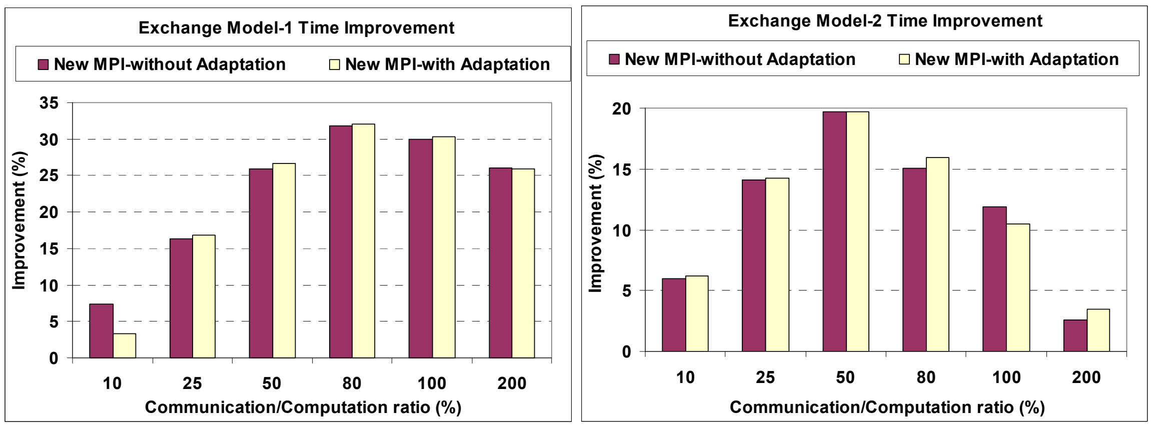


Figure 4.16 Exchange model micro-benchmarks: improvement over current Rendezvous protocol.

4.4.2 Application Results

In the previous section, I presented and analyzed the behavior of the proposed Rendezvous protocol with a number of micro-benchmarks, including the exchange model micro-benchmarks that can represent the communication core of some scientific applications. To see the effect of the proposed protocol on MPI applications, in this section I elaborate on experimentation with CG, BT, and LU application benchmarks from the NPB benchmark suite, and RADIX application. I have selected RADIX, and the above benchmarks from NPB, because they utilize MPI non-blocking calls. All of these applications have different communication characteristics.

I have run the NAS parallel benchmarks with class A and B, and the RADIX application with a larger workload to fit the messages into the Rendezvous range, on the 4-node cluster. I have measured the overall application wait time (including MPI_Wait, MPI_Waitall and

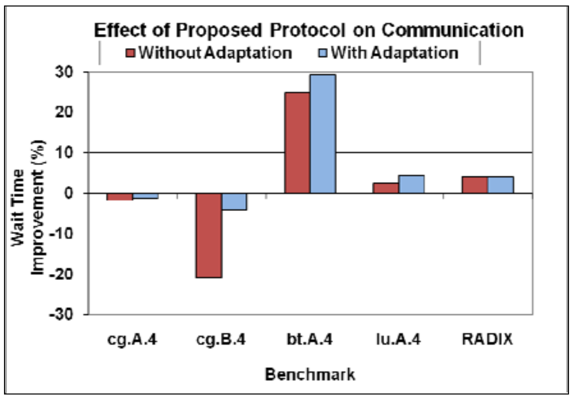
MPI_Waitany) for the current protocol and the proposed Rendezvous protocol (with and without adaptation). For the adaptation, I have used 80% RTR usage as the threshold for stopping/resuming RTR generation. Once the RTR usage of the process falls below this threshold, the RTR generation is stopped. Then the program silently monitors the cases that could potentially use RTR but they are not using it due to the adaptation mechanism being in effect. Once the silent RTR monitoring statistics reach the 80% threshold again, the RTR generation is resumed.

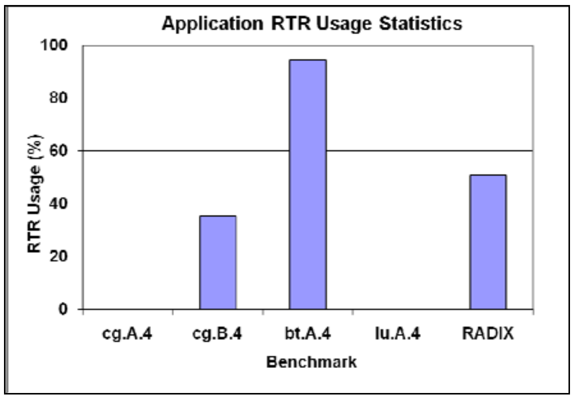
Figure 4.17(a) shows the improvement that the new protocol has made to the communication time of each application. Figure 4.17(b) shows the global RTR usage percentage for each application. The results presented are the average of five runs.

As shown in Figure 4.17, the communication time in some applications can significantly benefit from the protocol, while in others the wait time may increase to a great extent when adaptation is not used. For example, up to 30% reduction in the overall wait time for BT.A benchmark can be observed. BT.A heavily uses the Rendezvous protocol with a very high RTR usage percentage. Adaptation slightly affects the BT.A benchmark. However, LU.A is a benchmark that cannot use the proposed protocol because it is using *MPI_ANY_SOURCE* in its receive calls. The slight (~5%) improvement seen for LU.A is mostly due to the impact of the early progress engine call. The RADIX application is also gaining around 4% improvement, mostly because the communication processing in back-to-back communication calls is overlapped more with computation phases, due to using both RTR and initial progress engine call in Rendezvous communications.

On the other hand, CG is negatively affected by the proposed protocol. CG.A, which does not have any Rendezvous communication is just affected by a small overhead (less than 2%). On the other hand, CG.B that heavily uses the Rendezvous protocol is affected by 21% increase in wait time, when adaptation is not in use. Because the RTR usage percentage of CG.B falls below the threshold, the adaptation switches to the old Rendezvous protocol, reducing the overhead to only 4%, as shown in Figure 4.17.

Characteristics of the applications under study are the main reasons behind the reported results. Figure 4.18 briefly illustrates the computation/communication pattern for part of each application that is involved in non-blocking communications. For the CG.B benchmark, the code does not have any computation interleaved with the non-blocking communication. Therefore, there is not much opportunity for the application to utilize the receiver-initiated Rendezvous protocol for more overlap. This confirms the results presented in Figure 4.17.





(a) Wait time improvement **(b) Global RTR usage**

Figure 4.17 Effect of proposed Rendezvous protocol on applications communication time.

For BT.A benchmark, in each iteration, producing data for a non-blocking send call is overlapped with a non-blocking receive call. Similarly, LU.A has a communication pattern that overlaps with computation. Therefore, similar to what we observed in Section 4.4.2 with the exchange model benchmark, if the receive call is issued earlier, the new protocol receiver will initiate the Rendezvous (using RTR) and the non-blocking send call at the other side will start the data transfer. On the other hand, using the current Rendezvous protocol, the early receive call should wait until the MPI wait call to initiate the data transfer.

For the LU.A benchmark, despite interleaving computation with non-blocking calls, speculation cannot take place simply because the receiver does not know its peer process (recall the use of *MPI_ANY_SOURCE* in the receive calls).

The RADIX application has a high communication-to-computation ratio that is independent of the problem size and the number of processes [91]. Communication is performed in the form of a number of non-blocking receive calls, each four of them followed by an MPI Waitall call, and then followed by a number of MPI_Send calls. In some cases, the send phase precedes the receive phase. A simplified form of its communication pattern is shown in Figure 4.18.

CG.B		**BT.A**

Produce
Loop i
  MPI_Irecv(from process i)
  MPI_Send (to process i)
  MPI_Wait()
End Loop
Consume
		Produce($data_0$)
		MPI_Isend($data_0$)
		Loop i=1 to iterations
		  MPI_Irecv($data_i$)
		  Produce($data_i$)
		  MPI_Wait(i-1,Isend)
		  MPI_Wait(i,Irecv)
		  Consume($data_i$)
		  MPI_Isend($data_i$)
		End Loop

LU.A		**RADIX**

MPI_Irecv (any_source)
Produce (data)
MPI_Send (neighbor)
MPI_Wait()
Consume()
		Main Receive Loop:
		  Loop i = 1 to 4:
		    MPI_Irecv($Data_i$, Src)
		  MPI_Waitall()
		End Receive Loop
		Main Send Loop:
		  MPI_Send(Dst)
		End Send Loop

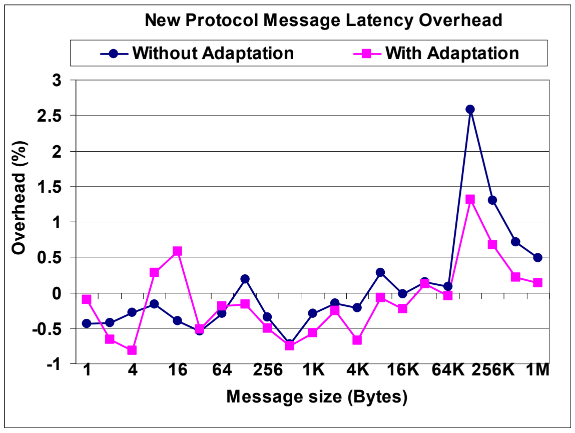
Figure 4.18 Non-blocking communication patterns for MPI applications

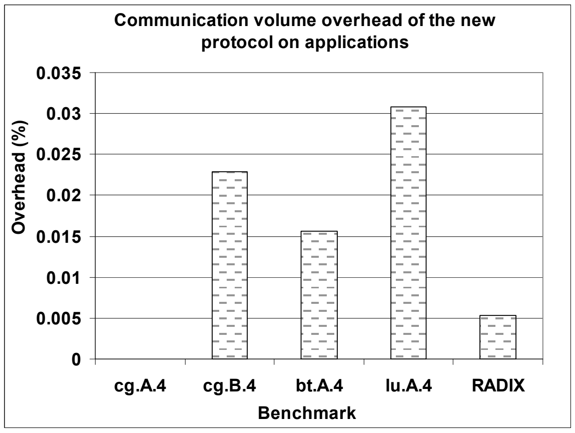
4.4.3 Communication Latency and Data Volume Overhead

As discussed earlier, the proposed Rendezvous protocol has some overhead associated with its design, especially when the adaptation mechanism is not in effect. Certain parts of the protocol that are used for race hazard prevention are the main sources of overhead. Examples are RTR and RTS acknowledgements, Eager-flag manipulation and extra Recvq searches. Other sources of overhead are the dropped RTR messages that will consume processing and communication resources at both ends of the network, though not benefitting the communication.

In this section, using a micro-benchmark I show the overhead of the new protocol on MPI message latency. I also present the amount of communication volume overhead for the applications, due to exchanging the newly introduced negotiation packets (RTR and ACK messages). For the message latency overhead, a basic ping-pong benchmark is used, with a timing that all Rendezvous receive calls will produce RTRs, but the RTRs will be dropped due to crossing with simultaneously generated RTSs from the peer process. The results for this benchmark are shown in Figure 4.19(a), presenting the overhead with and without the adaptation mechanism. No substantial overhead (less than 1%) is observed for Eager messages, while a small overhead (less than 3%) exists for Rendezvous messages, only if the adaptation is not in effect.

For the communication volume overhead, I count the amount of extra data generated due to the introduction of RTR and acknowledgement packets. I consider all dropped (unused) RTR messages and also all RTR-ACK and RTS-ACK messages as the sources of the new protocol's overhead. For each benchmark, the ratio of the extra data volume to the total application message volume is reported. As shown in Figure 4.19(b), less than 0.04% extra data is generated in the worst case among these benchmarks. These results confirm that the new protocol introduces a low level of overhead on latency and data volume.





(a) Latency overhead **(b) Communication volume overhead**

Figure 4.19 Overhead of the proposed protocol

4.5 Summary

To increase communication progress and overlap in MPI library over RDMA-enabled networks, in this chapter I proposed a novel speculative MPI Rendezvous protocol and implemented it on MPICH2 over NetEffect 10-Gigabit iWARP Ethernet. The experimental results show that the new protocol is able to effectively improve the receiver-side progress and overlap ability from almost zero to nearly 100%, at the expense of only 2-14% progress overhead at the sender-side, and less than 3% latency overhead, when the receiver arrives first. The implementation has also been able to improve the receiver-side overlap for more than 80%, and also make the receiver-side progress independent without reducing the sender-side overlap and progress ability. I also evaluated the proposal using exchange model micro-benchmarks. The evaluation confirms that the new protocol can contribute significantly to the performance of applications with this kind of core computation model. The results show that the amount of improvement depends on the ratio of communication to computation.

At the application level, I have compared the results using NAS benchmarks and the RADIX application. The results show up to 30% improvement in the application wait time. The results also show that the protocol may impose overhead on some applications due to their communication patterns and timing. To address this issue, I introduced a dynamic adaptation mechanism to switch back to the current Rendezvous protocol when the RTR usage statistics fall below a certain threshold. The adaptation mechanism has been quite successful in minimizing the amount of overhead on the applications.

Last but not least, I would like to add that although the 10-Gigabit iWARP Ethernet is used as the interconnect in this study, the proposed MPI Rendezvous protocol has implications beyond this network, and can be applied directly to any other RDMA-enabled networks such as InfiniBand.

Chapter 5

Improving MPI Small-Message Latency for Frequently-used Buffers

RDMA-based communication requires the source and destination buffers to be registered to avoid swapping memory buffers before the DMA engine can access them [59]. Memory registration is an expensive process that involves buffer pin-down and virtual-physical address translation [54, 59], as well as advertising the registration tag to the remote node. These costs lead MPI developers to treat small and large messages differently, in order to avoid extra communication overheads. For small user buffers, where the cost of buffer registration is prohibitive, the user data are copied from application buffers into pre-registered and pre-advertised intermediate buffers. These buffers are internal to the MPI implementation and are only used as temporary buffers for direct memory access at both send and receive sides. This is in contrast to large messages, where the overhead of the memory copy exceeds the memory registration cost, hence the actual user buffers should be registered for RDMA transfer.

The MPICH Eager protocol does not use an acknowledgment at the MPI level to notify the completion of the data transfer. The underlying network transport protocol used in this study (InfiniBand RC transport) is a reliable transport and the MPI send operation immediately finalizes the communication as soon as the data is sent to the other side over the IB network. At the other side, the receiver polls on the receive buffers for the message arrival.

On the other hand, in Rendezvous protocol used for large messages, after an initial negotiation, the data is directly transferred to the final application buffer at the receiver side and a finalization message is sent by the sender or receiver after the data transfer to inform the other side that the data is placed in its application buffer.

Similar mechanisms are used for MPI-2 one-sided operations (MPI_Get and MPI_Put), where a synchronization operation is required to finalize the communication. In both Put and Get operations, small messages are copied into intermediate pre-registered buffers, while the large application buffers are registered and used directly for RDMA transfer. For MPI_Put, once a message is transferred using an RDMA Write, a flag is set at the remote node (again, using an RDMA Write) to indicate that the data transfer is complete. In MPI_Get, the data is transferred using an RDMA Read into a local buffer, and then a done packet is sent to finalize the communication. In an active-target communication scenario [63], an explicit synchronization is performed among processes using RDMA Write operations; after all one-sided operations are posted.

In current MPICH/MVAPICH implementations, the aforementioned protocols for small messages are optimized based on a single usage of application buffers. In contrast, the reality is that most of the user buffers in MPI applications are frequently reused during the course of execution. This feature, buffer reuse in MPI applications, has been the main motivation behind this work.

To reduce the communication latency for small messages, I propose to register the frequently-used application buffers so that we can initiate RDMA operations directly from the application buffers rather than the intermediate buffers [84]. Obviously, infrequently-used buffers are treated as before, until they are marked as frequently-used. This way, the cost of communication is decreased by eliminating one data copy operation per data transfer. The registered user buffer can be kept in the MPI registration cache and be retrieved for subsequent references to the same buffer. Therefore, the cost of one-time registration is amortized over the cost of multiple data copies that are eliminated. Note that the receiver-side data copies for two-sided communications are still required, because we are carrying out an Eager transfer, which assumes no negotiation with the receiver process. For one-sided operations, however, only one data copy exists, a data copy at the sender-side for MPI_Put, or a data copy at the receiver-side for MPI_Get. This single copy operation is eliminated by the proposed method. Therefore, for one-sided operations we achieve true zero copy [77].

The proposed idea has been implemented in MVAPICH2 over InfiniBand. The results confirm that eliminating the data copy for buffers that are sufficiently reused can decrease the communication latency by up to 20% for small-message transfer protocols. The results also show that the method improves execution time of some HPC applications that frequently reuse their small message buffers. The method is made adaptive in a way that it is deactivated if the application does not benefit from it [84].

The rest of this chapter is organized as follows. In Section 5.1, I describe the motivation behind this work. I will then review some related work in Section 5.2. Details about the proposed method for small-message transfers will follow in Section 5.3. I also present an online adaptation mechanism to minimize the overhead for applications that do not reuse their buffers frequently. The experimental framework and evaluation results are presented in Section 5.4 and Section 5.5, respectively. Section 5.6 contains a summary of the chapter.

5.1 Motivations – Buffers Reuse Frequency

Despite architectural and technological advances in memory subsystems of modern computing systems, buffer copying operations are still a major source of overhead in inter-process communication. As explained earlier, the current protocols for small-message transfer use memory copies from the application buffer into intermediate buffers (or vice-versa), to avoid the higher cost of the memory pinning required for direct memory access by the network interface card. In such cases, if the user buffer is used frequently in an application, it may be beneficial to register the user buffer and avoid the memory copy. To find out whether this method is helpful, we first need to investigate the buffer reuse pattern of MPI applications at runtime.

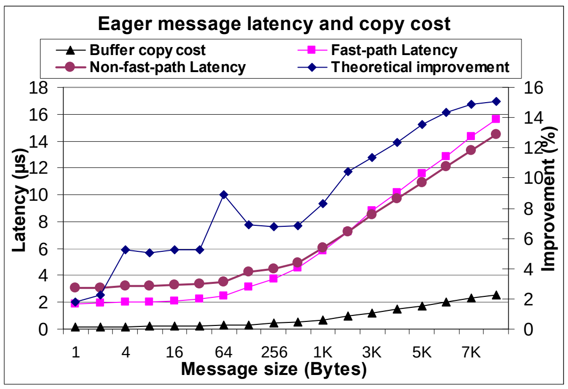
Table 5.1 shows the Eager buffer reuse statistics for some MPI applications described in Section 5.4.1. The second column shows the range (and the average) of the maximum number of times any user buffer is reused in a 16-process job. The third column shows the most frequently-used buffer sizes among 16 processes for these applications. Table 5.1 confirms that indeed application buffers are reused frequently in most processes of the applications under study. These observations confirm that there is a potential to revise the small message transfer protocols for the frequently-used buffers.

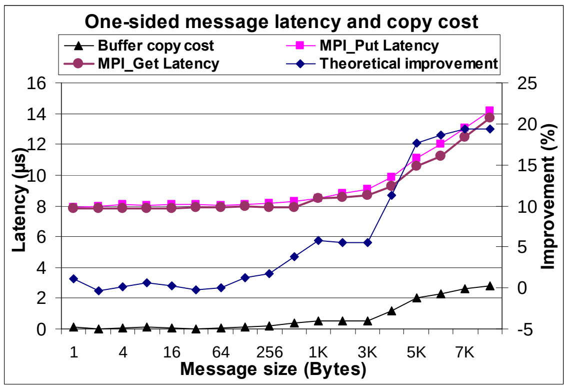
Table 5.1 Eager buffer reuse statistics for MPI applications running with 16 processes

MPI Application	Buffer reuse count	Most frequently-used buffer sizes (bytes)
NPB CG Class C	7904 - 7904 (Avg: 7904)	8 – 8 (Avg: 8)
NPB LU Class C	3749 - 3750 (Avg: 3749)	1560 - 1640 (Avg: 1600)
ASC AMG2006	45 - 292 (Avg: 119)	8 - 3648 (Avg: 770)
SPEC MPI2007 104.MILC	2 - 5010 (Avg: 2049)	8 - 4800 (Avg: 2876)

It is worthwhile to know how much improvement we can theoretically achieve if we eliminate one data copy operation. Figure 5.1 compares the ping-pong message latency against the cost of one buffer copy for small-size messages on our InfiniBand cluster using MVAPICH2 (refer to Section 5.3 for a complete description of the experimental platform). Figure 5.1(a) shows the results for two-sided communications while Figure 5.1(b) is for one-sided operations.

Performance results suggest that up to 20% latency improvement can be achieved in a ping-pong benchmark, by bypassing one buffer copy for small messages. Note that this analysis does not take into account the one-time cost of registration as well as the implementation overhead.





(a) two-sided operations (Eager) **(b) one-sided operations**

Figure 5.1 Small message latency, buffer copy cost and expected theoretical improvement.

As shown in Table 5.2, for two sided operations, MVAPICH2 switches from the Eager protocol to the Rendezvous protocol at about 9KB on our platform. Therefore, the experimental results in this chapter are shown for up to 8KB messages. I have not changed the default Eager/Rendezvous switching point in this study because it is the optimal value that the MVAPICH2 code calculates for the underlying platform. While a higher switching point is expected to provide a larger improvement for the proposed technique, it would degrade the baseline communication performance of the MVAPICH2 implementation.

For one-sided operations, this switching happens at 4KB on our platform. However, experiments show that we can get a better performance by increasing this switching point. For consistency, I will report the performance results for one-sided operations up to the 8KB messages, similar to the two-sided results.

Table 5.2 Small/Large message protocol switching points for one-sided and two-sided communications in MVAPICH2

Communication Model	Default Small/Large Switching Point	Modified Small/Large Switching Point
Two-sided (Eager protocol)	9KB	9KB
On-sided (Put and Get)	4KB	9KB

5.2 Related Work

There exists some related work on improving the performance of small-message transfers and I will point to some of the most important ones. In [33] a header cache mechanism is proposed for Eager messages in which the Eager message header is cached at the receiver side, and instead of a regular header a very small header is sent for subsequent messages with matching envelope.

Liu et al. [52] designed an RDMA-based Eager protocol with flow control over InfiniBand. If the RDMA flow control credits of a connection are used up without being released by the receiver, the communication falls back on the Send/Receive-based (two-sided) channel.

The authors in [93] describe a user-level pipeline protocol that overlaps the cost of memory registration with RDMA operations, which helps achieving good performance even in the cases of low buffer reuse. This method is for earlier versions of RDMA-enabled interconnects in which the registration cost was comparable to the cost of data transfer. The other contribution of this work is toward eliminating the need for a pin-cache in MPI, in order to avoid associated memory management complications.

Some work has also looked at the cost of memory registration in RDMA-enabled networks, especially its high costs for small buffers [20, 102]. In a recent work presented in [26], researchers have proposed a pinning model in Open-MX based on the decoupling of memory pinning from the application, as a step toward a reliable pinning cache in the kernel. Their proposal enables full overlap of memory pinning with communication. The pinning of each page can be overlapped with the communication of the previous memory page. This method works in Open-MX, which does not bypass the kernel. Such a method cannot be used in systems that are exclusively using user-level libraries and OS bypass.

In another work, the authors in [72] present an improved memory registration cache that performs memory region registration instead of individual buffer registrations. This method also uses batch de-registrations to reduce the overhead. They also propose a method to overlap data transfer with remote node registration. The work in [72] uses simulation to evaluate the proposed mechanisms for RDMA-based web applications.

In [22] the authors propose a number of memory-management techniques to improve memory registration issues in file servers. They pipeline subsequent file transfers over the network, in order to overlap data copies with data transfers. In another approach, they try to make the OS kernel ready for using the Linux *sendfile* mechanism for RDMA connections. This mechanism helps avoid file copies into the user space.

5.3 Proposal for Frequently-used Small-buffer Protocols

In this section, I describe the details of the proposed small-message communication mechanism, both for Eager protocol (two-sided) and one-sided operations. Section 5.3.1 describes the general behavior of the proposed Eager method. Section 5.3.2 shows the extension of this method for one-sided operations. Section 5.3.3 presents the details about detecting frequently-used buffers at runtime. Finally, Section 5.3.4 introduces an adaptation mechanism to minimize the overhead on applications that do not benefit from the new protocol due to their low buffer reuse statistics.

5.3.1 Enhanced Eager Protocol

Figure 5.2 and Figure 5.3 depict the general idea behind bypassing the user-buffer copy through application buffer registration. Figure 5.2 illustrates the two-sided (Eager) protocol. The current general path for Eager messages is shown in dashed arrows, while the new path for frequently-used buffers is in solid arrows. If an application buffer is used frequently, it will be registered so that the user data can be directly transferred from the application buffer by an RDMA operation. Therefore, the communication cost is reduced by avoiding the data copy into a pre-registered buffer. However, as noted earlier, we cannot avoid the copy into the receiver's intermediate buffer due to the Eager communication semantics.

In order for the receiver side to detect reception of Eager messages, there is a need to transfer the Eager message header along with the Eager payload. The Eager header constitutes a small amount of data (especially when header caching is enabled) that is still copied into the intermediate buffers. I use the InfiniBand RDMA scatter/gather mechanism to gather the header information from a pre-registered buffer and the user data from the application buffer, and then transfer them together into the receiver's pre-registered buffer. Using this mechanism, the registered application buffers are kept in the MVAPICH2 registration cache, so that they could be retrieved in subsequent references. MVAPICH2 uses a lazy de-registration mechanism to avoid re-registrations for future buffer reuses [60], as described in Section 3.5.1.

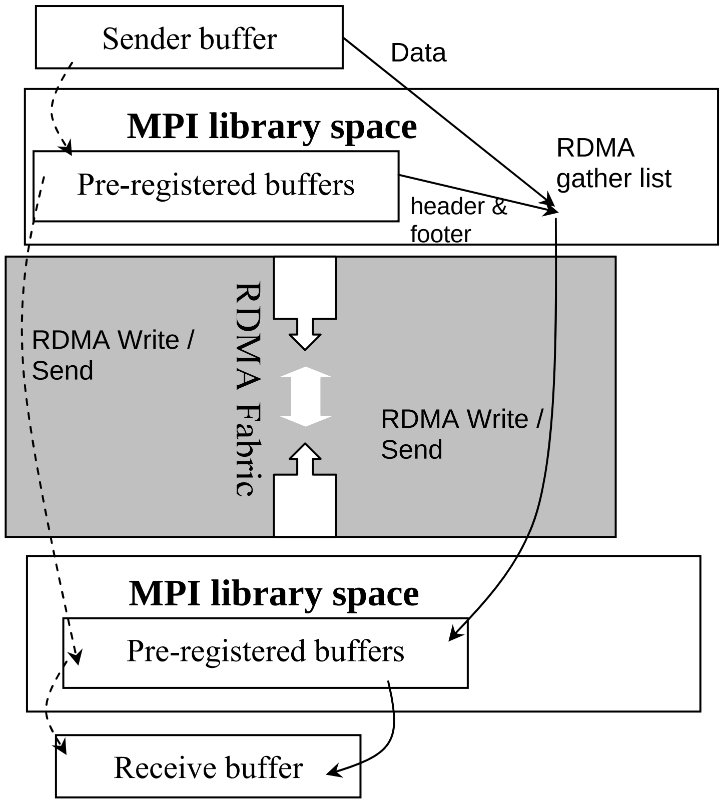


Figure 5.2 Data flow diagram for the current and proposed *Eager* protocols.

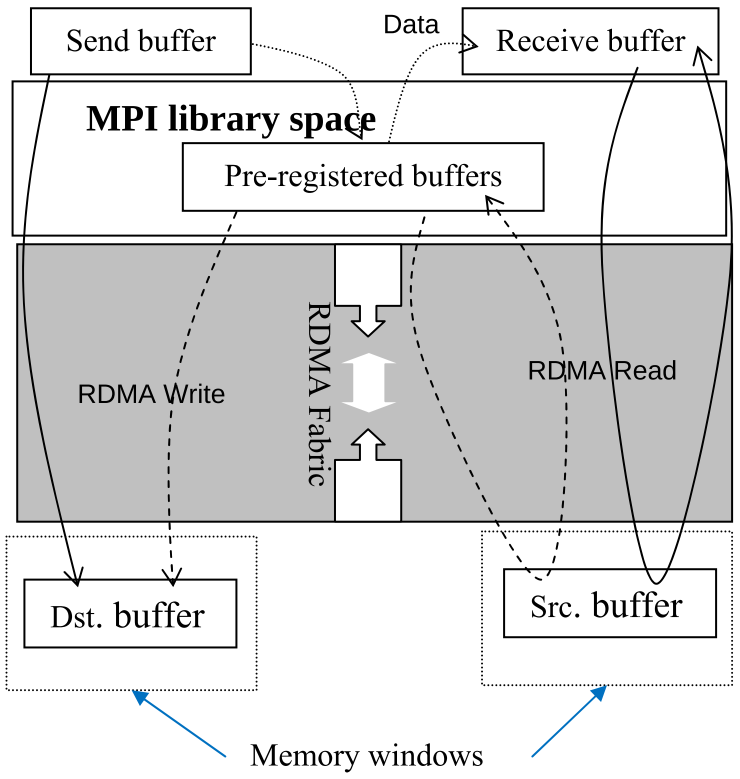


Figure 5.3 Data flow diagram for the current and proposed one-sided *put* protocol (left) and *get* protocol (right).

5.3.2 Enhanced One-sided Communication Protocol

Figure 5.3 shows the mechanism used for MPI one-sided *put* and *get* operations. MPI_Put is shown on the left hand side and MPI_Get is on the right hand side of Figure 5.3. Dashed lines show the current protocol, while the solid lines represent the new protocol in which a memory copy is bypassed. Similar to the Eager protocol for two-sided communications, if an application buffer (send buffer or receive buffer) is used frequently it will be registered for the data transfer in order to avoid a data copy into a pre-registered buffer. In both protocols, after initiating RDMA (Write or Read) operations, a remote flag is set using RDMA Write to inform the remote party that the operation is performed.

5.3.3 Detecting Frequently-used Buffers

To be able to detect frequently-used buffers, we need to keep track of the buffer usage statistics. When a small buffer is accessed for communication, a search is conducted for that buffer in a data structure containing buffer usage statistics. If the buffer is found in the table and its usage statistics have exceeded a pre-calculated threshold, then it is considered as a candidate for buffer registration. At this step, the algorithm looks for the buffer in the registration cache (for possible previous registration). A buffer that is not found in the cache will be registered and placed in the cache for future reference.

Now, the question is which buffer is considered frequently-used? In other words, what is the lowest value of the reuse threshold from which this mechanism can yield benefit? To find the answer, we need to calculate the timing costs associated with both communication paths depicted in Figure 5.2 and Figure 5.3.

Detection in Two-sided Communication: The current Eager communication path involves a copy to/from the pre-registered buffer at the sender/receiver, plus an RDMA Write or Send/Receive between the two pre-registered buffers. In the proposed technique, we do not have the first copy, but we have an extra memory registration at the sender side. If we consider C_m as the copy cost, NT_m as the network transfer time (or computation time, in case non-blocking calls are used to overlap communication with a computation phase), R_m as the registration cost for a single message with size m, and V as the implementation overhead for the new method, the current and the new communication times, T_c and T_n, when we reuse the buffer n times will be defined as:

$$T_c = n \times (2 \times C_m + NT_m) \qquad (5.1)$$

$$T_n = n \times (C_m + NT_m + V) + R_m \qquad (5.2)$$

In order to benefit from this method, we need to find the minimum n such that $T_n < T_c$:

$$n > \frac{R_m}{C_m - V} \qquad (5.3)$$

The value V is the overhead incurred in searching both the table containing buffer usage statistics and the buffer registration cache. The buffer usage statistics are stored in a hash table structure, and the registration cache is in a balanced binary tree.

Inequality (5.3) shows the minimum number of times a buffer of size m needs to be reused after registration in order to benefit from the new method. Table 5.3 shows the minimum value of n for different message sizes on the evaluation platform. The message sizes are chosen in 1KB steps. The fast-path results are for RDMA Write based communication while the non-fast-path results are for Send/Receive based communication. The value, n, is negative for messages smaller than 64 bytes for all cases, due to $V > C_m$. Even with very high number of reuses, there will be no benefit, and therefore the proposed method is disabled for such message sizes.

Table 5.3 Minimum required buffer reuse number, *n*, for different message sizes on our platform

Fast-path	Non-fast-path	One-sided
Message size (bytes): Minimum *n*	**Message size (bytes): Minimum *n***	**Message size (bytes): Minimum *n***
< 1KB: 432	< 1KB: 1568	< 1KB: 196
1KB: 200	1KB: 448	1KB: 128
2KB: 110	2KB: 224	2KB: 128
3KB: 76	3KB: 131	3KB: 125
4KB: 60	4KB: 95	4KB: 56
5KB: 48	5KB: 75	5KB: 33
6KB: 40	6KB: 65	6KB: 29
7KB: 36	7KB: 55	7KB: 26
8KB: 32	8KB: 49	8KB: 25

The new algorithm dynamically decides when a buffer needs to be registered. The registration threshold is based on the pre-calculated values, shown in Table 5.3. However, I have devised the algorithm so that it speculatively decides to register a buffer when it is reused at least by a certain portion (e.g. 25%, 50%, or 100%) of the minimum number, *n*, hoping that the buffer will be reused for more than that later. As an example for the 25% case (used in the experiments in this chapter), a 4KB buffer is registered when it is reused 15 times, hoping that it will be reused 60 times or more so that the registration cost is amortized.

Detection in One-sided Communication: MPI one-sided communication follows a different path. For MPI_Put, a memory copy is followed by an RDMA Write operation to the remote buffer exposed by MPI window. Thus, the communication cost for the current and new methods, when a buffer is reused *n* times**,** can be formulated as in Equation (5.4) and Equation (5.5), respectively**,** in which NW_m is the latency of a network write (RDMA Write) operation:

$$T_c = n \times (C_m + NW_m) \qquad (5.4)$$

$$T_n = n \times (NW_m + V) + R_m \qquad (5.5)$$

For $T_n < T_c$, *n* follows the same inequality (5.3). A similar analysis can be done for MPI_Get, in which the RDMA Read cost (NR_m) is used instead of NW_m in Equation (5.4) and Equation (5.5), leading to the same lower bound for *n*. Table 5.3 shows the minimum value of *n* for different message sizes for one-sided communication on our platform.

Searching the Buffer-usage Table: In order to minimize the overhead, *V*, I have experimented with two different data structures for the buffer-usage table: a hash table and a balanced binary tree [60]. The hash table has 1M hash buckets, and each buffer address is hashed into a 20-bit index. In each hash bucket, the buffers with conflicting hash values are stored in a linked list. For the hashing function, I have chosen a multiplicative hash method proposed for the

Linux buffer cache [49] that is known to be efficient with a low conflict rate for hashing buffer addresses.

As shown in Figure 5.4, with the growth of the database (the number of small-size buffers in the table), the search time in the hash table grows very slowly, compared to that of the binary tree. Based on these results, I have chosen the hash table for the buffer-usage search structure.

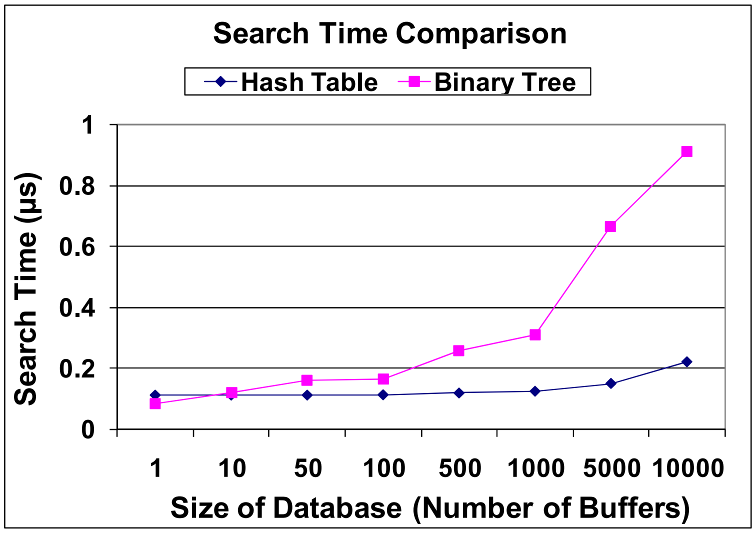


Figure 5.4 Comparing search time between hash table and binary tree.

5.3.4 Adaptation Mechanism

The idea of registering small buffers is useful for applications with a high buffer-reuse profile. For applications with low buffer reuse, we normally do not incur the buffer registration cost for small messages. However, the overhead of manipulating buffer usage statistics can affect the overall performance of the communication protocol, especially for small messages with latencies comparable to the overhead.

One approach is to obtain static profiles for the applications and use them to register frequently-used buffers when the program starts running. However, this approach requires extra provisioning, especially at the compiler stage, for application profiling. In addition, it does not work for applications with dynamic profiles. The approach I have used in this work is based on monitoring application buffer usage at runtime. The overall buffer-usage statistics of the application are dynamically monitored, and the growth of the buffer-usage table is adaptively stopped when the overall buffer-usage statistics are low.

To avoid early stoppage, the adaptation mechanism acts only when the hash table starts growing linked lists in its hash buckets. This is because the overhead of hash table search/insertion increases dramatically when the hash values for message buffers conflict and need to be added to the same hash bucket linked list, causing the linked-list to grow. Obviously, if no conflict has occurred in the hash-table search, the search time is of order O(1) regardless of the number of buffers in the hash table. In summary, adding more buffers to the hash table is stopped when the following two conditions are satisfied:

(1) When the number of (registered) buffers in the hash table that are marked as frequently-used is less than 20% (a typical value) of all buffers in the table; and

(2) When the hash table has started to grow linked lists (due to linear search/insertion costs).

5.4 Experimental Framework

The experiments are conducted using four Dell PowerEdge 2850 servers, each with two dual-core 2.8GHz Intel Xeon EM64T processors (2MB of L2 cache per core) and 4GB of DDR-2 SDRAM. Each node has a Mellanox ConnectX DDR InfiniBand HCA installed on an x8 PCIe slot, interconnected through a Mellanox 24-port Infiniscale-III switch. In terms of software, MVAPICH2 1.0.3 is used over OFED 1.4, installed on Linux Fedora Core 5, kernel version 2.6.20.

5.4.1 MPI Applications

Four applications are used to evaluate the proposed Eager protocol: CG and LU benchmarks from NPB 2.4 benchmarks [67], AMG2006 from ASC Sequoia suite [48], and 104.MILC from SPEC-MPI 2007 suite [96].

Details about CG and LU can be found in Section 4.3.1. AMG2006 is a parallel algebraic multi-grid solver for linear systems arising from problems on unstructured grids. It uses blocking and non-blocking MPI send/receive calls and collective communication calls such as broadcast, gather, scatter, all-reduce, all-to-all, and all-gather.

SPEC 104.MILC simulates four-dimensional SU(3) lattice gauge theory. The benchmark is for the conjugate gradient calculation of quark propagators in quantum dynamics. It uses non-blocking MPI send/receive calls and some broadcast and all-reduce collective calls.

5.5 Experimental Results

In this section, I present the performance results of the proposed small-message transfer protocols using point-to-point, one-sided, and collective communication micro-benchmarks, as well as some MPI applications.

5.5.1 Micro-benchmark Results

Two-sided communication: Two micro-benchmarks are used to evaluate the potential of the proposed method in improving MPI two-sided communications (Eager protocol):

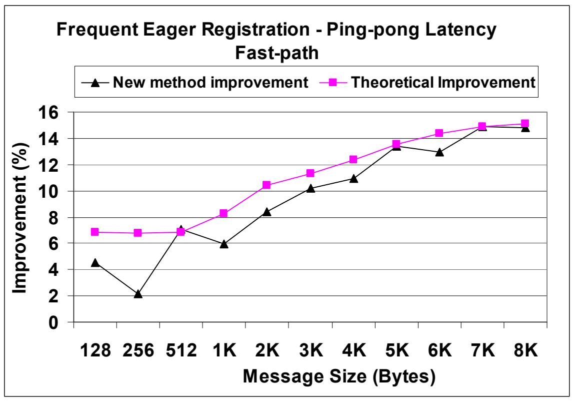
1) Ping-pong Latency: The ping-pong operation is repeated 10,000 times and the average one-way latency is measured for different messages in the Eager message communication range, from 128 bytes to 8KB. The send and receive buffers are maintained separately to avoid affecting each other in terms of memory access.

Figure 5.5 shows the amount of improvement in the Eager message latency when the sender-side copy is removed and the user buffer is registered for RDMA transfer. As stated earlier, the new method is disabled for messages smaller than 128 bytes due to registration cost and implementation overhead that makes buffers smaller than 128 bytes inappropriate for this method, although the theoretical value of n for 128-byte messages is positive. The maximum improvement is around 14% for 8KB messages, very close to the 15% theoretical improvement discussed in Section 5.1.

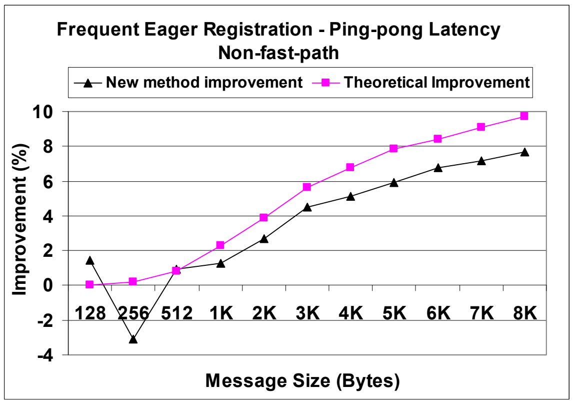
Note that the theoretical improvement is calculated from reducing the communication latency by the cost of one buffer copy minus the implementation overhead for the new method, as shown in Equation (5.6):

$$Theoretical\ improvement = (Copy\_overhead - New\_overhead)/Old\_Latency \quad (5.6)$$

Obviously, this calculation does not take into account the presumably amortized one-time overhead of buffer registration. In calculating the theoretical improvement, I use the buffer copy cost incurred in the actual micro-benchmark test, rather than using a separate micro-benchmark for measuring the copy cost. This is because the buffer copy cost varies in different situations, depending on the alignment of both source and destination addresses, with respect to CPU word size and memory page size [2, 61]. That is why in the send/receive based micro-benchmark (non-fast-path), as shown in Figure 5.5(b), the improvement (for both theoretical and actual cases) is lower than the RDMA-based fast-path one. The investigation shows that the memory copy cost in the former case is lower.



(a) Fast-path

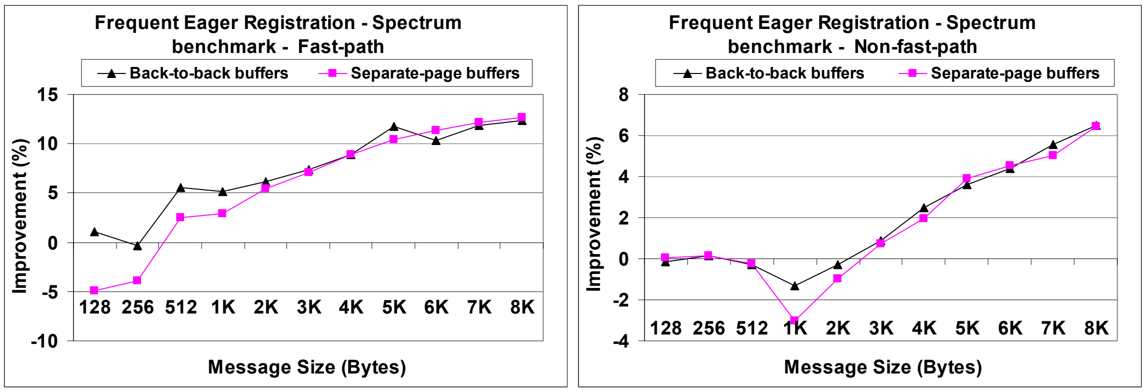


(b) Non-fast-path

Figure 5.5 Message latency improvement for (a) fast path and (b) non-fast-path.

2) Spectrum Buffer Reuse Latency: The ping-pong test examined the case with maximum buffer-reuse percentage and high reuse numbers. To simulate an application with different buffer-reuse patterns for different buffers, the next micro-benchmark uses a spectrum of buffer-reuse patterns by increasing the reuse count of each buffer from 1 to 1000. I have considered a 1000-unit buffer set: $\{ buf_i \mid 1 < i < 1000 \}$, in which buf_i is being reused i times.

For efficiency and integration purposes, the MVAPICH2 registration mechanism and registration cache are directly used. MVAPICH2 is registering application buffers in one-page chunks (4KB on our system). Thus, two buffer allocation schemes are chosen: in one case, the buffers are allocated back to back in memory. In the other case, the buffers are allocated in separate memory pages, without any page overlap. In the case that the buffers are allocated back to back, registration of one buffer may result in registration of a page that is overlapped with the next buffer, slightly decreasing the registration cost for the next buffer. Thus, the separate-page allocation case essentially shows the worst-case scenario for buffer registration cost.



(a) Fast-path (b) Non-fast-path

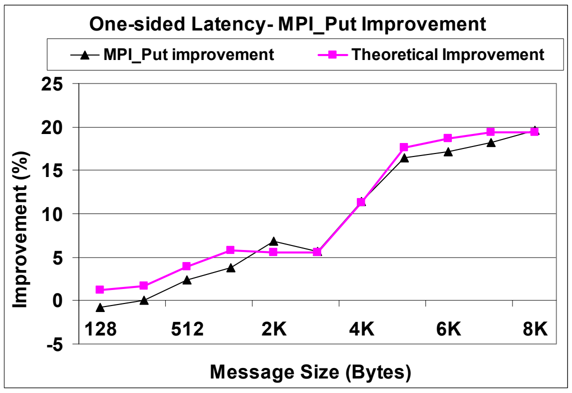
Figure 5.6 Improvement as observed in spectrum buffer usage benchmark for (a) fast path and (b) non-fast-path.

The results for this micro-benchmark using both buffer-allocation schemes are shown in Figure 5.6. The effect of the buffer-allocation scheme is evident for smaller messages, but almost vanishes as messages approach the page size and beyond. The highest improvement from using the new method (12.7%) is again for 8KB messages.

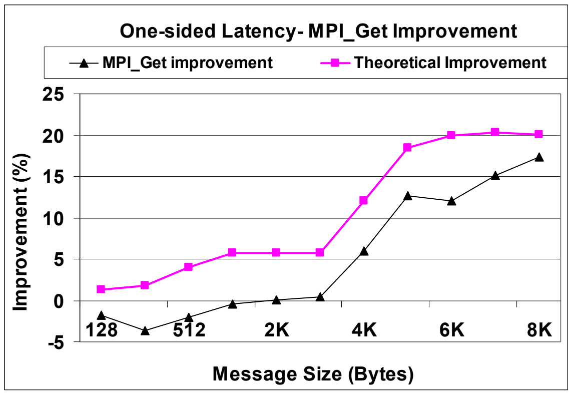
Effect on One-sided Communications: In this section, I present the results for two similar micro-benchmarks as for the two-sided communication, but using one-sided communication primitives (i.e. MPI_Get and MPI_Put operations). In both tests, the application buffers are allocated in separate memory pages to avoid overlap of registration cost.

In the first test, each one-sided communication call (MPI_Get or MPI_Put) is followed by a one-sided active-target synchronization (i.e. MPI_Fence). The loop is repeated 10,000 times and the average time is measured. One-sided synchronization in the micro-benchmark is required, because the MPI implementation performs one-sided operations in the subsequent synchronization calls. Figure 5.7 presents the latency results of this micro-benchmark. The results are compared to the theoretical improvement, calculated using Equation (5.6). The MPI_Put actual improvement results are very close to the theoretical expectation. The MPI_Put latency is improved for messages larger than 256 bytes and reaches around 20% for 8KB messages.

On the other hand, the MPI_Get improvement is noticeably lower than the expected calculated amount and is evident only after 3KB buffer size. My investigation into the MPI_Get code reveals that since the eliminated buffer copy in MPI_Get is performed in the last stage (after receiving data from the source using an RDMA Read) in MPI_Fence, part of it is overlapped with the previously-issued RDMA Write operations used to finalize the synchronization process. Therefore, the buffer copy overhead on the original MPI_Get communication time is less than the actual memory copy cost, which was used for the estimation of theoretical improvement.



(a) MPI_Put



(b) MPI_Get

Figure 5.7 MPI one-sided latency improvement for (a) MPI_Put and (b) MPI_Get.

The second micro-benchmark studied is similar to the spectrum micro-benchmark used for two-sided communication. This test is used to estimate the effect of the proposed method on benchmarks with a spectrum of buffer-reuse patterns. One-sided operations are called in the same way as the first micro-benchmark. The improvement results are presented in Figure 5.8. Obviously, the improvement is lower than the previous micro-benchmark, because not all buffers are being reused 100% of the time.

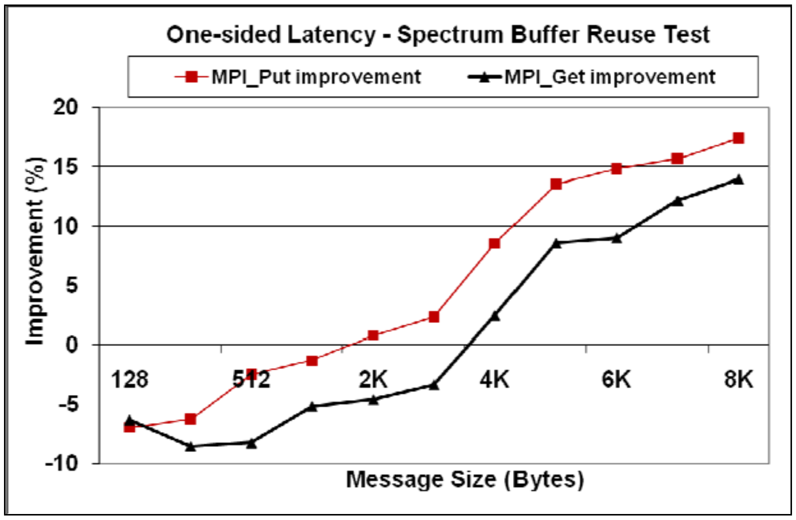


Figure 5.8 MPI One-sided improvement as observed in spectrum buffer usage benchmark.

Effect on Collective Communications: Most of the MPI collective operations use MPI point-to-point primitives (i.e. MPI_Send, MPI_Recv, and MPI_SendRecv) to implement the collective algorithm. Therefore, improving the performance of point-to-point communications can indirectly affect the collective performance as well. In this section, I present the effect of our Eager protocol improvement on MPI broadcast and scatter collective operations.

In each iteration of the micro-benchmark, all processes are engaged in the collective operation, followed by a synchronization operation (MPI_Barrier). The test is repeated 10,000 times and the average time of the collective operation is calculated across all processes. The synchronization is used to prevent process skew from propagating in subsequent iterations of the micro-benchmark.

Figure 5.9 shows the performance improvement for MPI_Broadcast and MPI_Scatter operations running with four processes and 16 processes on our 4-node cluster. Broadcast and Scatter use the binomial tree algorithm for data transfer in the range of messages under study. The only difference is that scatter distributes data from distinct but back-to-back buffers, while broadcast distributes data from a single source buffer among processes.

The results for the 4-process cases are relatively good, even better than the point-to-point results (for broadcast), because more than one data copy is eliminated per collective operation. In the 16-process case, intra-node communications are done through shared memory, and therefore such communications will not use the improved communication path, resulting in a lower improvement percentage compared to the 4-process cases. In addition, the scatter operation uses many intermediate buffers that are not re-used, and that is why its 16-process improvement is the lowest.

The curves for the scatter operation sharply decrease for messages larger than 4KB. The reason is that in the first subdivision step of the binomial tree algorithm, the root transfers half of the data (at least two buffers) to the first intermediate node. Therefore, the data size exceeds the Eager/Rendezvous switching point (9KB) for messages larger than 4.5KB. The other factor is related to the use of temporary buffers in intermediate transfers. Thus, the effect of the proposed algorithm is reduced for scattering messages larger than 4.5KB.

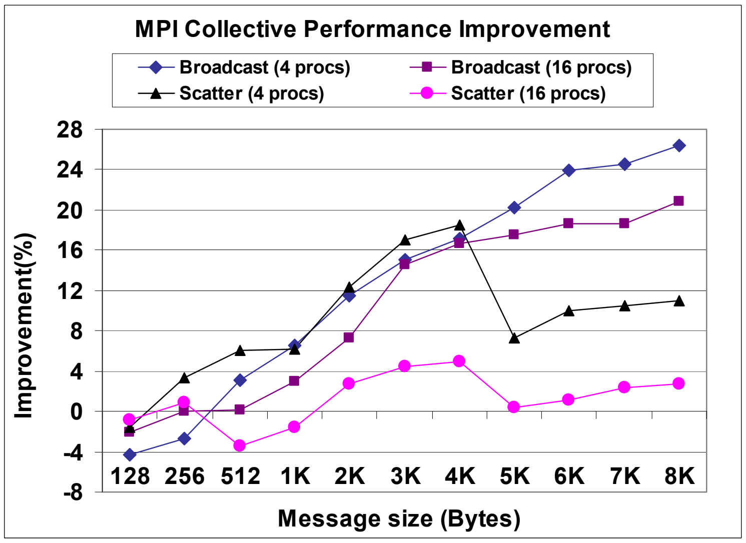


Figure 5.9 Improvement of MPI collective operations.

5.5.2 Effect on MPI Applications

In this section, the effect of the proposed method on some MPI applications is evaluated. These applications are chosen for this study because they have different buffer-reuse characteristics. In the evaluation, three values are measured for each application: the amount of time spent in MPI send operations; the total communication time spent in MPI send, MPI receive, and MPI wait operations; and the application execution time. Figure 5.10 shows the improvements achieved using the proposed method. Table 5.4 shows the frequently-used buffer percentage for the applications. These values show the ratio of the frequently-used registered buffers over all Eager buffers in the buffer reuse hash table.

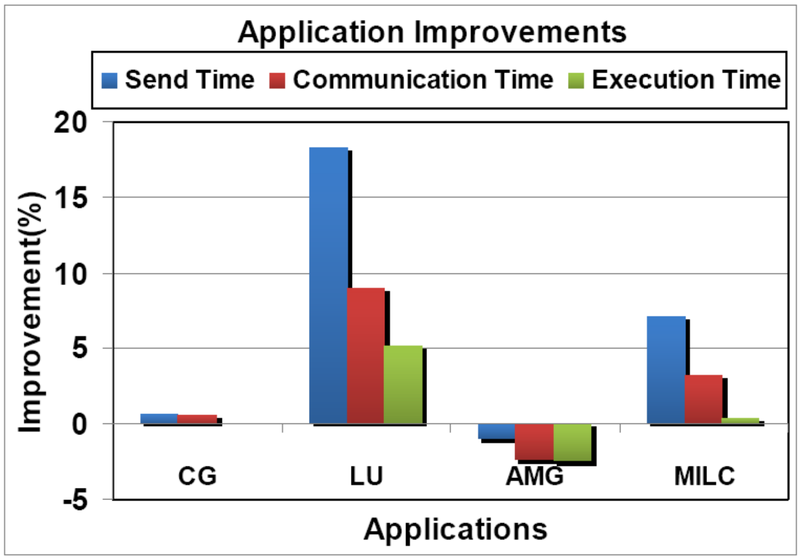


Figure 5.10 MPI application improvement results.

Table 5.4 Frequently-used buffer percentage for MPI applications (for messages > 128B)

MPI Application	Most frequently-reused buffer percentage
NPB CG Class C	0%
NPB LU Class C	32.9%
ASC AMG2006	8.7%
SPEC MPI2007 104.MILC	22.6%

Both MILC and LU have a high buffer reuse profile. That is why they benefit from the proposed technique. Their send times have 7% and 18.3% improvement, respectively. Obviously, not all of this gain translates into total communication-time improvement. In fact, there are some general factors affecting the overall gain in total communication time and application runtime, as summarized below.

- The level of synchronization among processes and the time that a matching receiver is called at the receiver is one important factor. In case of skewed receivers, the send-time improvement will vanish during the skew time.
- The ratio of total communication time to the application runtime is another factor that affects the total gain. Applications with lower communication/computation time ratio will see a smaller portion of the communication time gain reflected into the application runtime.

For example, I believe that process skew is the major cause for this in LU application, since its processes only synchronize once at the beginning.

CG and AMG2006 are applications with lower buffer-reuse statistics. The new method has a slight effect on their performance (less than 1% for CG and around 2% for AMG). Although the CG buffer-reuse statistics shown in Table 5.1 are very high, those are only for small messages (8 bytes). Since the method is disabled for messages smaller than 128 bytes, the new technique is effectively disabled for CG.

The reason that AMG2006 does not gain is that its buffer-reuse statistics are very close to the speculative threshold point at which the algorithm starts to register buffers. However, as stated before, those threshold values are chosen to be 25% of the minimum required reuse statistics (in Table 5.3). Thus, AMG2006 suffers from the overhead of registering insufficiently reused buffers. AMG2006 also has a low buffer-reuse ratio (Table 5.4) that is lower than the minimum for our adaptation (20%). Therefore, the adaptation is activated, preventing the buffer-reuse table from growing. This reduces the overhead on AMG. For example in the experiments, if the adaptation is disabled, the total communication time of AMG increases by 6%, while with adaptation this increase is only about 2%.

5.6 Summary

In this chapter, a novel technique has been proposed to improve the MPI small-message transfer protocols over RDMA-enabled networks. In the proposed method that addresses both two-sided and one-sided communications in MPI implementation, one buffer copy is eliminated and the application buffer is used to transfer the data directly. The proposed technique is suited for applications with high buffer-reuse statistics. The adaptation mechanism minimizes the overhead on applications with low buffer reuse profiles.

Micro-benchmark results show that the new method achieves up to 14% improvement in the point-to-point Eager communication time, using RDMA Write (fast-path) operation. This is very close to the maximum theoretical improvement of about 15% on our platform. The improvement for the Eager protocol using Send/Receive operations (non-fast-path) is less, but still close to its own theoretical maximum.

Micro-benchmark results for MPI one-sided operations also show significant improvement (close to 20% for MPI_Put and approximately 17% for MPI_ Get operations). I have also shown that collective communications such as broadcast can achieve greater improvements (up to 25%) because more than one data copy is eliminated in each operation.

MPI application results confirm that applications with high buffer reuse benefit from this technique. Such applications show much higher improvement for the send-time than the total communication time and application execution time. One should note that the process synchronization pattern and the time ratio of communication to computation are the deciding factors that may affect the improvement in the total communication time and application runtime.

Chapter 6

Extending iWARP-Ethernet for Scalable Connection-less Communication over HPC Clusters

Due to easy deployment and low cost of ownership, Ethernet is ubiquitously used in commercial and research clusters, for both HPC and datacenter services. Despite its performance inefficiencies, Ethernet currently accounts for more than half of the interconnection networks in the top 500 supercomputers [100].

As discussed in Chapter 2, critical CPU availability and performance issues caused by the large overhead of Gigabit and 10-Gigabit Ethernet network protocol processing [25] has initiated a wide range of efforts to boost Ethernet efficiency, especially targeting its latency for HPC. The first major attempt was offloading TCP/IP processing using stateless offload (e.g. offloading checksum, segmentation and reassembly, etc.) and stateful TOE [25].

Another major approach on top of TOE has been equipping Ethernet with techniques such as RDMA and zero-copy communications, which have traditionally been associated with other high-performance interconnects such as InfiniBand. iWARP [85] was the first standardized protocol to integrate such features into Ethernet, effectively reducing Ethernet latency and increasing host CPU availability by taking advantage of RDMA, kernel bypass capabilities, zero copy, and non-interrupt-based asynchronous communication [6, 80]. Rather than the traditional kernel-level socket API, iWARP provides a user-level interface that can be used in both Local Area Network (LAN) and Wide Area Network (WAN) environments, thus, efficiently bypassing kernel overheads such as data copies, synchronization, and context switching [30].

Despite its contributions to improving Ethernet efficiency, the current specification of iWARP lacks functionality to support the whole spectrum of Ethernet-based applications. The current iWARP standard is only defined on top of reliable connection-oriented transports. Such a protocol suffers from scalability issues in large-scale applications due to memory requirements associated with multiple inter-process connections. In addition, some applications and data services do not require the reliability overhead and implementation complexity and cost associated with connection-oriented transports such as TCP.

In this chapter, I propose to extend the iWARP standard on top of UDP in order to utilize the inherent scalability, low implementation cost, and minimal overhead of datagram protocols[1]. This proposal provides guidelines and discusses the required extensions to different layers of the current iWARP standard in order to support the connection-less UDP transport. It is designed to co-exist with the current standard while being consistent and compatible with its current connection-oriented specification. While the proposed extension is designed to be used in datacenter and HPC clusters, the emphasis of this work is on HPC applications.

The implementation of datagram-iWARP in software reveals that significant performance benefits can be potentially achieved when using datagrams in iWARP-based Ethernet clusters. For instance, the micro-benchmark results show up to 25% small-message latency reduction and up to 40% medium- and large-message bandwidth improvement when using MPI on top of datagram-iWARP. It is observed that MPI applications can substantially benefit performance and memory usage when running on datagram-iWARP, compared to the connection-based iWARP.

[1] This work has been done in collaboration with my colleague, Ryan Grant. The theoretical part in Section 6.3 has been developed by me, with intellectual inputs from Ryan Grant. The MPI work in Section 6.4.2 as well as the OF verbs development in Section 6.4.1 have been done by me. The rest of the work in 6.4.1 regarding the integration of datagram transport in software iWARP has been done in close collaboration with Ryan, with the native verbs implementation developed by him. I would also like to thank him for putting a lot of time and effort in helping to gather the results presented in Section 6.5 and Section 6.6.

Up to 30% less memory usage and up to 45% runtime reduction are achieved for some HPC applications on a 64-core cluster.

The rest of this chapter is organized as follows. In Section 6.1, I discuss the shortcomings of the current iWARP standard, primarily for HPC clusters. Section 6.2 discusses some scholarly work aimed at improving the Ethernet protocol in different directions. In Section 6.3, guidelines for changes to the iWARP standard for datagram support are proposed. Section 6.4 describes the implementation of iWARP over UDP. Section 6.5 and Section 6.6 include the experimental platform and evaluation results respectively. Finally, Section 6.7 provides a summary of the chapter and points to future works in this direction.

6.1 Shortcomings of the Current iWARP Standard

The current iWARP standard offers a range of capabilities that increase the efficiency of Ethernet in modern HPC and datacenters clusters. Taking advantage of the well-known reliable transports in the TCP/IP protocol suite is one of the key advantages of the iWARP. Reliability has in fact been a major force for designing the current iWARP standard on top of connection-oriented transports. The LLP in iWARP is assumed to be a point-to-point reliable stream, established prior to any communication in the iWARP mode. This requirement makes it easy for the ULP to assume reliable communication of user data. In addition, the independence of individual streams makes iWARP able to enforce error management on a per-stream basis.

Such a standard is suitable for applications that require strict reliability at the lower layer, including data validation, flow control, and in-order delivery. Examples for such applications are reliable datacenter services such as database services, file servers, financial applications, and policy enforcement systems (e.g. security applications, etc.). On the other hand, there is a growing demand for applications such as voice and media streaming that find the strict connection-based semantics of iWARP unnecessary. For such cases, the current iWARP standard imposes barriers to application scalability in large systems, due to its explicit connection oriented nature and reliability measures. The following subsections point to the shortcomings of the current standard and their relevant implications. As such, there are strong motivations for extending the iWARP standard with datagram transport.

6.1.1 Memory Usage Scalability

The scale of high-performance clusters is increasing rapidly and can soon reach a million cores. The pervasiveness of Ethernet in HPC clusters places a huge demand on the scalability of the iWARP standard that is designed to act as the main Ethernet efficiency solution.

An obvious drawback of connection-oriented iWARP is the connection memory usage that can grow by $O(n^2)$ of the number of processes. This dramatically increases the application's memory footprint, unveiling serious scalability issues for large-scale applications. As the number of required connections increases, memory usage grows proportionally at different network stack layers. In a software implementation of iWARP at the TCP/IP layer, each connection will require the allocation of a set of socket buffers, in addition to the data structure required to maintain the connection state information. Although the socket buffers are not required in a hardware implementation of iWARP due to zero-copy on-the-fly processing of data, making many connections will have other adverse effects. Due to limited RNIC cache capacity for connection state information, maintaining out-of-cache connections will require extra memory requests by the RNIC.

The other major source of memory usage is the application layer. Specifically, communication libraries such as MPI pre-allocate memory buffers for each connection to be used for fast buffering and communication management [44]. Scalability issues require changes to the standard to make it viable and effective for a large-scale cluster with millions of communicating processes.

6.1.2 Performance Barriers

Connection-oriented protocols such as TCP limit performance with their inherent flow-control and congestion management [35]. TCP slow start and congestion-avoidance adaptation mechanisms are more suitable for wide-area lossy networks that require reliable data service. HPC applications running on a local cluster do not require the complexities of TCP flow and congestion management. For such cases, UDP offers a lighte-weight protocol that can significantly reduce the latency of individual messages, closing the latency gap between iWARP and other high-speed interconnects. In addition, many datacenter applications such as multimedia applications using media streaming protocols over WANs can easily tolerate packet loss to some extent. These protocols are currently running on top of unreliable datagram transports such as UDP. Due to such semantic discrepancies, the current connection-oriented specification of iWARP makes it impossible for such applications to take advantage of iWARP's major benefits such as zero copy and kernel bypass.

6.1.3 Fabrication Cost

The complexities associated with stream-based LLPs such as TCP and SCTP translate into expensive and non-scalable hardware implementations. This becomes especially important with modern multi-core systems where multiple processes could utilize the offloaded stack. A heavyweight protocol such as SCTP or even TCP can partially support multiple parallel on-node requests, due to implementation costs associated with hardware-level parallelism [80]. This means that a small portion of many cores available on the node will be able to simultaneously utilize the actual hardware on the NIC. This can easily lead to serialization of the communication.

6.1.4 Hardware-level Multicast and Broadcast

iWARP lacks useful operations such as hardware-level multicast and broadcast. These operations, if supported, can be utilized in applications with MPI collectives and also media streaming services. iWARP does not support such operations primarily because the underlying TCP protocol is not able to handle multicast and broadcast operations. An extension of iWARP to datagrams will boost iWARP's position as a leading solution for high-performance Ethernet.

6.2 Related Work

The first hardware implementation of iWARP appeared in the Ammasso Gigabit Ethernet NIC [19]. NetEffect introduced a 10-Gigabit iWARP RNIC that works on the PCIe I/O interconnect [8]. The introduction of this RNIC was an important step in Ethernet latency improvement, putting an end to the high latency of Ethernet networks with an unprecedented 6μs user-level latency.

To reduce the expenses related to iWARP RNICs at the client side, and at the same time, take advantage of iWARP technology for servers, software implementations of iWARP are also available. Kernel- and user-space software implementations of iWARP have been reported in [21] and [18] respectively, to be used in client machines to connect to iWARP-compatible servers without the added cost for iWARP hardware. There is also a comprehensive software iWARP project currently under development, which is meant to be included in the OFED package [58]. Although this approach will not help reducing latency or increasing bandwidth, it helps reducing the server load by using an RNIC at the server side and software iWARP at the clients. To the best of my knowledge, the proposal in this chapter is the first and only work in this area that extends the iWARP standard to datagram transport and utilizes it for HPC applications.

Besides iWARP, there have been other approaches aimed at improving Ethernet efficiency for clusters using the transport layer of other high-performance interconnects over the Ethernet link layer. The first attempt was MXoE [64] designed by Myricom to provide the high-performance functionality of the MX library on top of an Ethernet network. Researchers at INRIA in France, in collaboration with Myricom prepared an open-source implementation of MXoE, called Open-MX [27]. There has been similar work on InfiniBand, such as InfiniBand over Ethernet (IBoE) that is also called RDMA over Ethernet (RDMAoE) [97]. RDMAoE is designed to take advantage of InfiniBand's efficient RDMA stack while simply replacing InfiniBand's link layer with Ethernet. This technology encapsulates InfiniBand reliable and unreliable services inside Ethernet frames. An evaluation of IBoE can be found in [97].

A new set of standards referred to as Converged Enhanced Ethernet (CEE) provide advanced features over Ethernet networks. Some industry vendors and researchers propose to include RDMA functionality over CEE (RoCEE) [16].

There has been some past work on the InfiniBand network to equip MPI with the UD transport for scalability purposes. In the MVAPICH-UD project [45], a UD-based channel is designed which has shown considerable memory usage benefits over the RC-based channel. MVAPICH Aptus is a continuation of the UD-based MPI over InfiniBand in [45].

6.3 Proposal for Datagram-iWARP

The current iWARP standard and its main layers (RDMAP and DDP) are explicitly designed for connection-oriented LLPs. Therefore, there are semantic discrepancies with datagram protocols such as UDP that need to be addressed in our design. There are features in the standard that make the definition of unreliable and datagram services viable in the current iWARP framework (for example, Sections 3.2 and 8.2 of the DDP specification [90]).

This proposal tries to keep the current well-developed specification of iWARP, while extending its functionality to support datagram traffic. As the first step, I highlight parts of iWARP at different layers that are incompatible with datagram semantics. Then, I propose guidelines to address such incompatibilities and include datagram semantics. In this research, I cover the untagged model of the DDP layer; the tagged model will be designed and implemented in future. The proposals in this chapter should not be considered as exact modifications to the standard, as such modifications are outside the scope of this research. I rather point to major places of the standard that need to be modified to support datagram transport. Figure 6.1 presents the major changes required, separated by different layers of the current standard. Details of the proposed changes can be found in a categorized form in the subsequent sections.

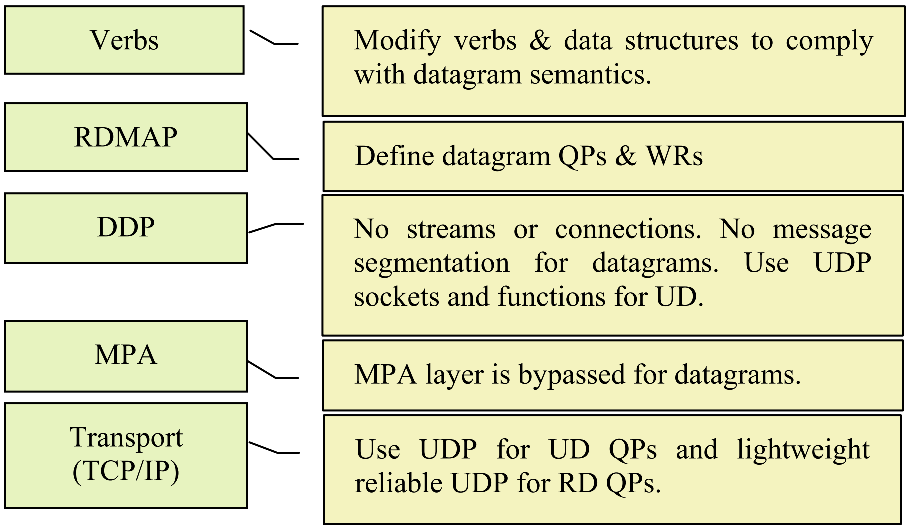

Verbs
Modify verbs & data structures to comply with datagram semantics.
RDMAP
Define datagram QPs & WRs
DDP
No streams or connections. No message segmentation for datagrams. Use UDP sockets and functions for UD.
MPA
MPA layer is bypassed for datagrams.
Transport (TCP/IP)
Use UDP for UD QPs and lightweight reliable UDP for RD QPs.

Figure 6.1 Extensions to the iWARP stack for datagram-iWARP.

6.3.1 Modifications to the Verbs Layer

To support datagram mode, it is not necessarily required to introduce new verbs. The existing set of iWARP verbs can be adapted to accept datagram related input and act according to the datagram service. Here, I point to some major parts of the verbs specification that need to be changed to support datagram transport:

- Currently, there is only one type of QP in iWARP, the connection-based QP. Thus, there has been no need for QP type definition in the standard. With the new extension, however, new QP type(s) must be added to distinguish between datagram-iWARP and connection-based iWARP. More details will be discussed in Section 6.3.2.

- QP creation and its input modifiers need to be changed. For example, to specify the transport type (connected or datagram) a new input modifier should be added to the QP attribute structure.
- Specification of the QP modify verb needs to change, to accommodate the new definition of the QP states and the required input data for datagram QPs. As an example, the datagram QPs need a pre-established datagram socket to be passed to modify the QP into the RTS state [31].
- An address handle is required for each send WR posted to the datagram QP to specify the receiver's IP address and UDP port related to the remote QP.
- Completion notification data structure needs to be changed to accommodate the new WR structure. In particular, the work completion structure should be changed to include the source address.

6.3.2 Reliability and In-order Delivery

Reliable service is a fundamental assumption in the current iWARP standard. This assumption is not necessarily in opposition to the use of datagrams. In datagram-iWARP design, two types of datagram services are introduced: UD, for which the reliability requirements of the current standard need to be relaxed, and RD, which can be built upon the current reliable iWARP. Subsequently, QP types need to be defined at the verbs and RDMAP layers. The defined QP types are: UD and RD for UDP and RC for the current TCP-based QPs.

The datagram-iWARP over UD transport assumes no reliability or order of delivery from the LLP. In the untagged model which is the focus of this work, the incoming messages will be matched to the posted receive WRs at the data sink, in order of their arrival at the DDP layer which is not necessarily their order of issue at the data source. While keeping the iWARP data integrity checksum mechanism (e.g. Cyclic Redundancy Code or CRC), the remaining reliability measures are left to the application protocol. Such a service is very useful for applications with high error resiliency (such as media-streaming applications in datacenters) and applications that can efficiently provide their own required level of reliability. An example is applications running in low error-rate environments such as closed Local or System Area Networks where standard reliability measures impose too much performance overhead.

For the RD service, the LLP is assumed to guarantee that messages from a specific data source are delivered correctly and in order. Such a definition implies a logical pseudo-stream between the local QP and the remote QP. However, DDP/RDMAP layers are not required to keep state information for such a logical stream. Similar to UD, for RD service, DDP and RDMAP layers are required to pass the messages in the order they have received them. The way the LLP (here, a reliable form of UDP) provides reliability and the mechanism by which such a service is configured are outside of the scope of the iWARP standard. To keep the scalability advantages of using a connection-less transport, the LLP reliability service is assumed to be lightweight and should require little or no buffering for individual remote sockets.

6.3.3 Streams and Connections

Currently, the RDMAP and DDP layer streams are assigned to underlying LLP connections that are assumed to be pre-established. Since connections are conceptually not supported in a datagram (connection-less) model, no connection establishment and teardown is required. For datagram-iWARP, a previously created UDP socket is required for each QP. In this case, the ULP transitions the QP into iWARP mode after creating the QP and assigning the lower-layer socket. This operation is done locally without negotiating any parameters (such as CRC settings) with any other peer. Such parameters need to be pre-negotiated by the ULP. This also implies that the ULP is no longer required to configure both sides for iWARP at the same time.

If a reliable datagram service is being used (i.e. RD mode), transport error management and connection teardown/termination requirements in the current standard will be the responsibility of the datagram LLP. Error management at the higher layers (e.g. DDP or RDMAP) needs to be modified to suit the datagram service. For example, the current standard requires an abortive termination of a stream at all layers and an abortive error must be surfaced to the ULP, should an error be detected at the RDMAP layer for that stream [87]. Since such a requirement does not apply to datagrams, an abortive error must be surfaced to the ULP, and the QP must simply go into the Error state without requiring any stream termination. This makes the QP unable to communicate with any other pair, until the QP is reset and modified to the RTS state by the application. Instead of the stream termination message sent to the other side, a simple error message should be transferred only to the peer process for which this error has happened, identifying the erroneous message number using the *Message Sequence Number* field (MSN in the DDP header). The error message can be placed into an Error queue that replaces the Terminate queue of the connected mode.

6.3.4 QP Identification Number

For the untagged model, the DDP layer provides a *queue number* (QN) in the DDP header to identify the destination QP [90], which is currently not fully utilized by the RDMAP. The RDMAP only uses its first 2 bits [87]. In the datagram-iWARP, we currently assume assignment of a single datagram QP to a UDP socket. In such a model, no QN is required to identify the source and destination QPs. An optional model of the datagram service can assign multiple datagram QPs to a single socket, similar to multiple streams per LLP connection in the current iWARP. Such a case, if included in the standard, can utilize the QN field in the DDP header.

6.3.5 Message Segmentation

Unlike TCP byte-oriented service, UDP datagrams will arrive at the LLP in their entirety, if they arrive without an error. Thus the concept of message segmentation and out-of-order placement at the DDP layer is irrelevant to the datagram service. This implies that the DDP layer does not need its provisions for segmented message arrival over the datagram transport (including *Message Offset* (MO) and even MSN for some cases). For messages larger than the maximum UDP datagram size (64KB), segmentation at the application layer is required.

6.3.6 Completion of Work Requests

In the connected mode, a WR is considered complete when the LLP layer can guarantee its reliable delivery. The same semantics can be used for RD transport. However, for the UD transport, we no longer require an LLP guarantee. Thus, a WR should be considered complete as soon as it is accepted by the LLP for delivery.

6.3.7 Removing the MPA Layer

Since each DDP message will be encapsulated into one UDP datagram, no markers are required for iWARP over UDP. Therefore, the MPA layer (in particular the marker functionality) is not needed for the datagram service. This will improve the performance of the datagram transport, since MPA processing has shown to impose significant overhead on the performance of iWARP implementations due to complexities associated with marker placement [6], in addition to increasing the overall size of the required data transmission.

6.4 Software Implementation

To evaluate the proposed datagram extension to the iWARP standard, a software implementation of datagram-iWARP has been developed. Figure 6.2 shows the layered stack of this implementation. The implementation can be used on top of both reliable and unreliable UDP protocols. The evaluation in this chapter is on top of unreliable (regular) UDP.

To assess the implementation in a standard way, an OF verbs interface has been prepared on top of the native software iWARP verbs. The OF verbs interface has also been used to adapt an existing MPI implementation on top of our iWARP stack. The next subsections discuss some features of the implementation at both iWARP and MPI levels.

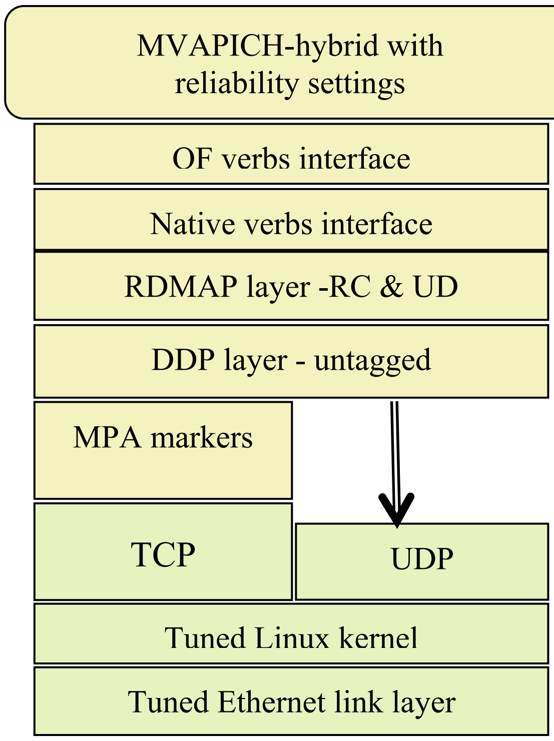


Figure 6.2 The software implementation of datagram-iWARP.

6.4.1 Software Datagram-iWARP

The software iWARP implementation from OSC [18] has been used as the code base and has been extended to the datagram domain. Here, I list a number of features of the implementation:

- Complete implementation and integration of iWARP over UDP, into the TCP-based iWARP stack from OSC. This has been done by introducing new native verbs to support datagram semantics.
- Using CRC at the lower DDP layer for the datagram integrity check.
- Using a round-robin polling method on operating sockets, to ensure a fair service to all QPs in the software RNIC. In this method, every time the device is polled, the next un-polled socket is polled first.
- Using I/O vectors for UDP communication (similar to TCP) to avoid extra sender- and receiver-side copies, for improved performance and CPU availability. In I/O vector calls (sendmsg and recvmsg), the message data and header can be gathered/scattered from/to non-contiguous data to/from the datagram socket buffer. Therefore an intermediate copy is not required to make the UDP datagram.
- Avoiding segmentation of DDP messages into MTU-size datagrams. This option, which is possible due to the message-oriented nature of UDP, positively contributes to the performance of datagram-iWARP.
- Implementation of standard OF verbs: These verbs were originally designed for InfiniBand (called OpenIB verbs). Currently, they are known as OF verbs and are utilized to implement iWARP verbs abstraction as well. The native verbs are used to implement the OF verbs that are utilized at the MPI layer. The evaluations show a small implementation overhead, between zero to about 10%, when using OF verbs on top of native verbs.

6.4.2 MPI over Software Datagram-iWARP

To evaluate the performance and memory usage of a datagram-based iWARP for HPC applications, an MVAPICH implementation has been adapted on top of OF verbs over software iWARP. Specifically, the hybrid channel from MVAPICH-Aptus over InfiniBand is utilized for this purpose. MVAPICH-Aptus is an available MPI implementation that offers a hybrid (UD and RC) channel over OF verbs. The hybrid channel offers dynamic channel management over InfiniBand's UD and RC transports, meant to offer scalability for MPI applications on large-scale InfiniBand clusters. The channel starts with UD-based QPs for each process, and based on a set of policies, establishes RC connections to a selected set of other processes, up to a pre-set maximum number of RC QPs for each process. This strategy makes applications scale better by limiting the resource-greedy RC connections and putting most of the communication on UD QPs. Reliability has been added for UD communication at the MPI layer, using acknowledgments and timeouts [44].

The MVAPICH code has been modified in several ways to adapt it over the iWARP standard and the software iWARP implementation. A list of some modifications I have made to the MVAPICH is presented below.

- Transforming MVAPICH UD-based connection management to the datagram-iWARP: This includes establishing datagram sockets and relevant address handles to be used as the underlying LLP (UDP) sockets required by datagram-iWARP.
- Transforming MVAPICH InfiniBand RC-based connection management: The MVAPICH hybrid channel uses on-demand RC connection management [44]. Due to semantic differences between TCP and InfiniBand RC, the handshaking steps for MVAPICH dynamic connection establishment have been modified. The new arrangement also piggybacks some new required information. In addition, it performs the socket connections at the very last handshake stage.
- Changing or disabling parts of the code relying on incompatibilities between iWARP and InfiniBand. This includes functions unsupported in the iWARP implementation such as immediate data, the DDP tagged model, Global Routing Header (GRH), SRQ, XRC, Service Levels (SL), Local IDs (LIDs), and LID mask control.

6.5 Experimental Framework

Two different clusters are used for the experiments. Cluster C1 is a set of four nodes, each with two quad-core 2GHz AMD Opteron processors, 8GB RAM, 512KB L2 cache per core and 8MB shared L3 cache per processor chip. The nodes are interconnected through NetEffect NE020 10GE cards connected to a Fujitsu 10GE switch. The OS is Fedora 12 (kernel 2.6.31). Cluster C2 contains 16 nodes, each with two dual-core 2.8GHz Opteron processors, with 1MB L2 cache per core, 4GB RAM and a Myricom 10GE adapter connected to a Fulcrum 10GE switch. The OS is Ubuntu with kernel version 2.6.27. The reason for using two clusters for the evaluation of this work is to show how performance and memory usage scale using datagram iWARP on two different architectures. In particular, the number of cores per node for C1 and C2 is different. With C2 having half of the C1 core per node ratio and twice the number of cores in total, its inter-node communication share will be four time that of the C1. This is expected to yield more performance and scalability improvements, since the proposed extension only affects MPI inter-node communications.

6.6 Experimental Results and Analysis

I assess the performance of the proposed UD-based datagram-iWARP using verbs and MPI level micro-benchmarks on both C1 and C2 clusters. I also evaluate the effect of datagram-iWARP on the performance and memory of some MPI applications on C1 and C2 clusters.

6.6.1 Micro-benchmark Performance Results

Micro-benchmarks are developed to test the performance of the UDP-based iWARP compared to that of the standard TCP-based iWARP. Figure 6.3 shows the OF verbs layer ping-pong latency results for both C1 and C2 clusters. It can be clearly observed that in most cases the UD latency is lower than that of the RC, primarily due to the following reasons:

- Communication over UDP offers a lighter and consequently faster network processing path.
- UD path is bypassing the MPA layer markers, while RC-based messages require markers, which are a significant source of overhead on all message sizes.
- The closed dedicated cluster provides an almost error-free environment where strict reliability measures of the TCP protocol are considered purely overhead compared to the unreliable UDP.

The reason for UD latency being significantly more than RC latency at 64KB is that it is exceeding the maximum datagram size and the benchmark needs to split the messages into 64KB chunks.

At the MPI layer, micro-benchmark results for latency and bandwidth are presented. For all datagram tests we use the MPI-level reliability provisions that exist in MVAPICH code for the UD transport [44]. These provisions include sequence numbers, acknowledgements sent for every 50 messages, and timeouts (a fraction of a second) in case acknowledgements are not received or out-of-sequence packets are received.

Figure 6.4 includes the ping-pong latency comparison of MPI over datagram and connection-based iWARP. The results that are presented for both C1 and C2 clusters similarly show the superiority of UD-based MPI performance over RC mode. The benefits can mostly be attributed to similar reasons as discussed above for the verbs performance results.

Figure 6.5 shows the bidirectional bandwidth results at the MPI level. For this test, I use two pairs of processes on two nodes, communicating in the opposite directions. In each pair, one of the processes posts a window of non-blocking receive calls. The other process in the pair posts a window of non-blocking send calls. Synchronization then occurs at the end.

As observed, MPI-UD offers a higher bidirectional bandwidth for most of the message sizes, meaning that we can better saturate the network using datagrams. The improvement reaches more than 40% for 2KB messages. Lighter protocol processing and minimal reliability measures are the advantages of UD-based communication that make the benchmark capable of pushing more data on the wire in each direction.

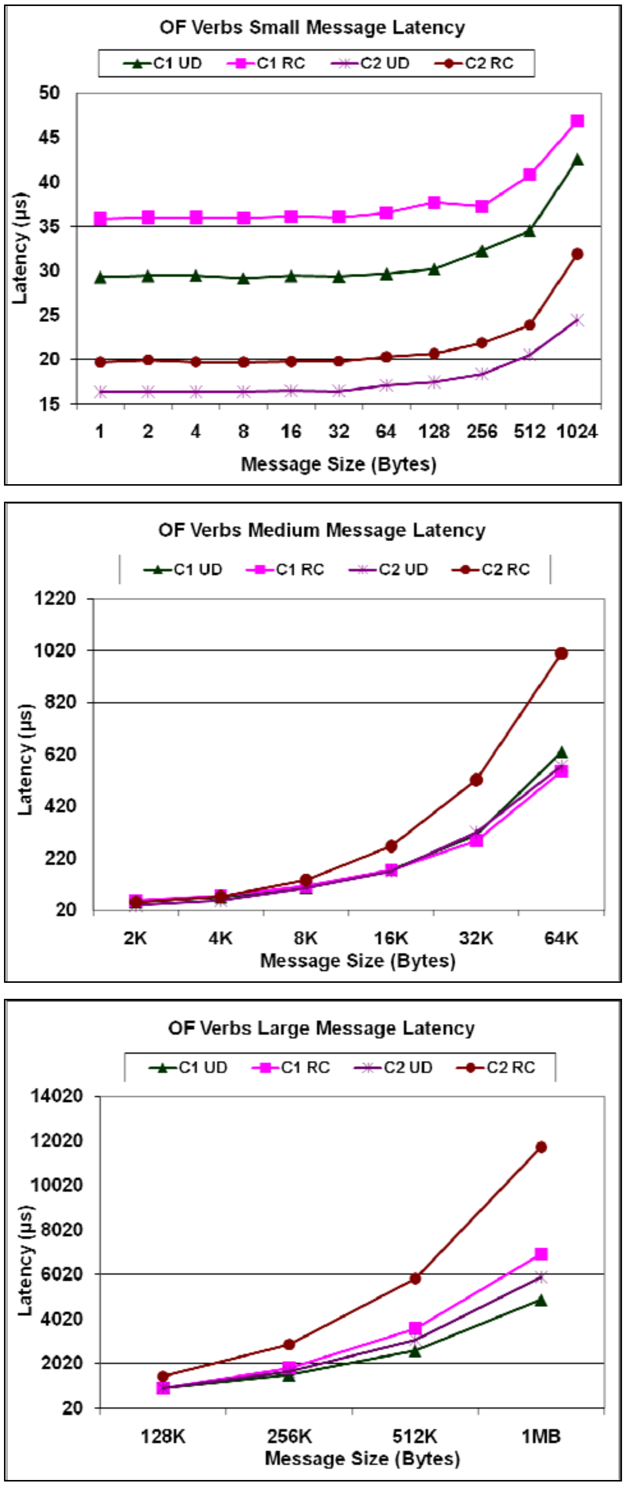
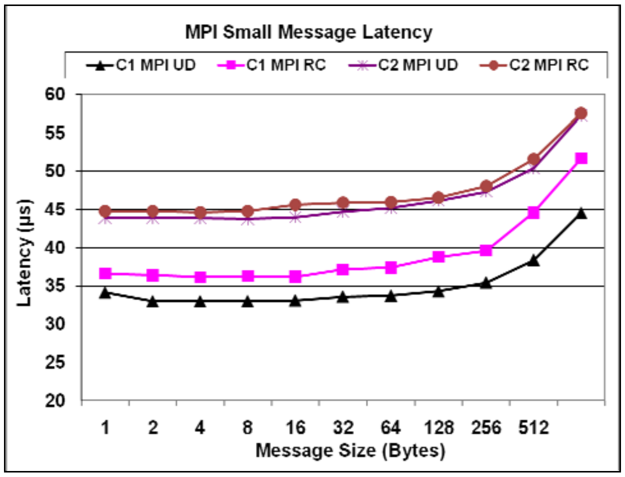
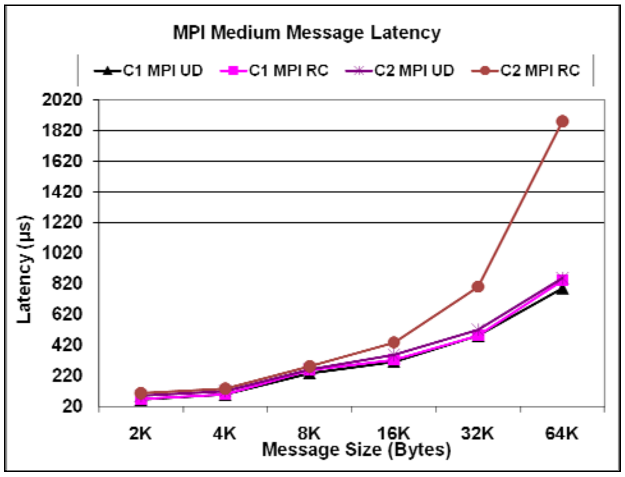


Figure 6.3 Verbs ping-pong latency.





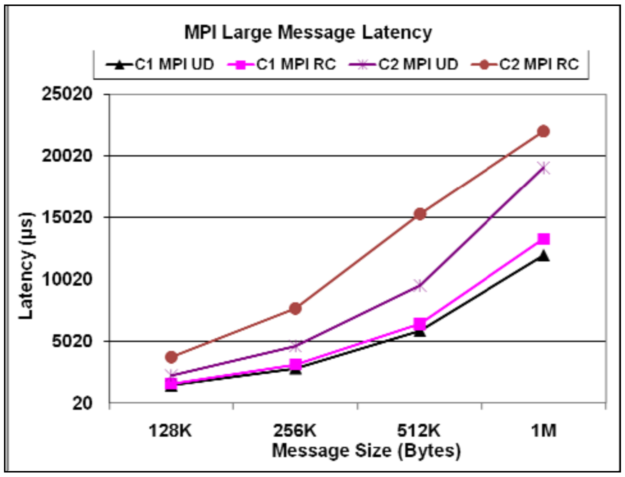


Figure 6.4 MPI ping-pong latency.

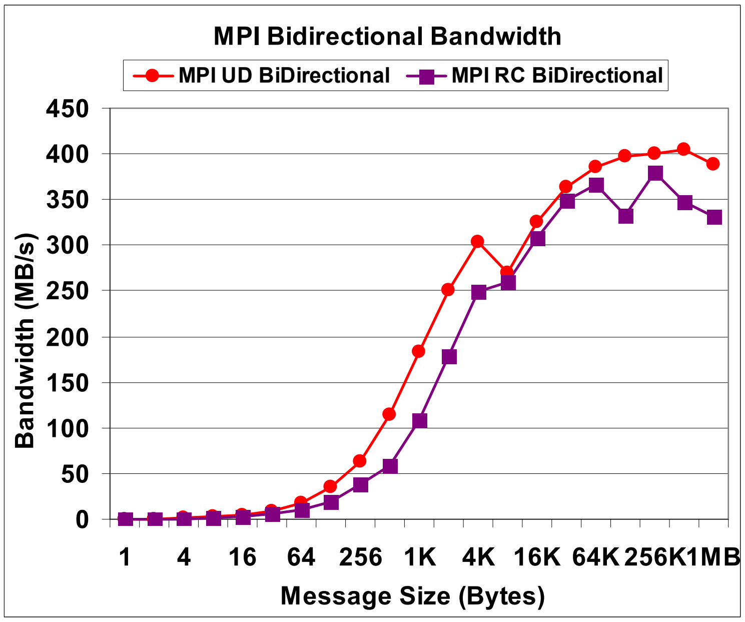


Figure 6.5 MPI bidirectional bandwidth.

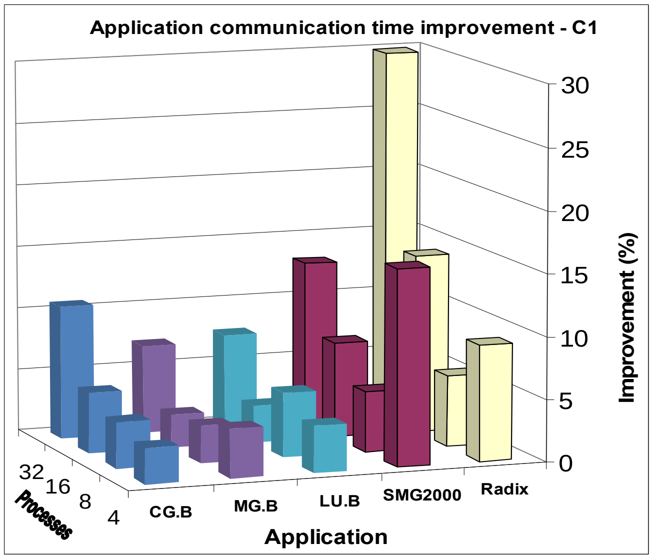
6.6.2 MPI Application Results

Application Performance

Figure 6.6 and Figure 6.7 present MPI application performance results, including total communication time and application runtime, respectively. For measuring communication time, the time spent in communication primitives (MPI blocking and non-blocking send and receive and MPI wait calls) are aggregated. In each figure, I have presented the results for both C1 and C2 clusters.

The results are reported for class B of CG, MG and LU benchmarks from the NPB suite version 2.4 [67], as well as the RADIX sort [91] and SMG2000 [12] applications. All results are presented for 4, 8, 16, 32, and 64 processes (64-process results are only on C2 cluster).

The results for both communication time and application runtime clearly show that one can expect considerable performance benefits when using datagram-based communications. In addition to reasons for superiority of datagrams discussed above, the lowered complexity of UDP should theoretically create the availability of more CPU cycles for the applications' computation phases, which would lead to lower overall runtimes.



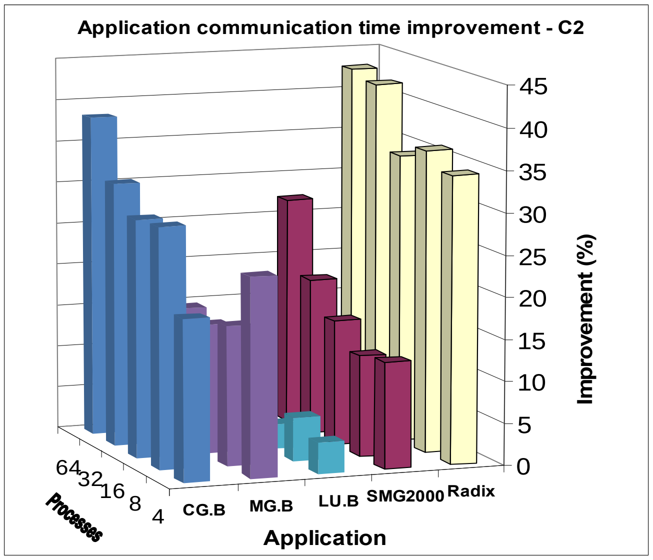
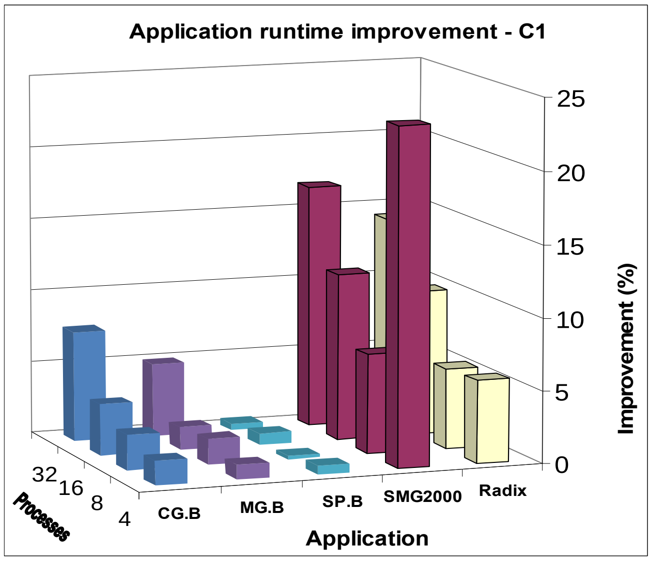


Figure 6.6 MPI application communication time benefits.



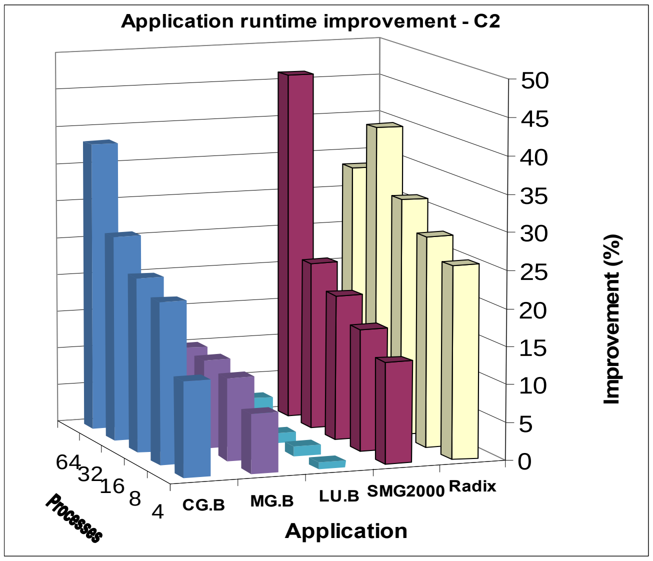


Figure 6.7 MPI application runtime benefits.

Application Memory Usage

As discussed in Section 6.1.1, for the connection-based model, the memory usage grows at different communication layers. One of the strongest motivations to extend the iWARP standard to the datagram domain is to address the O(n^2) increase in memory usage over connection-oriented transports, in order to make the standard scalable for large scale parallel jobs.

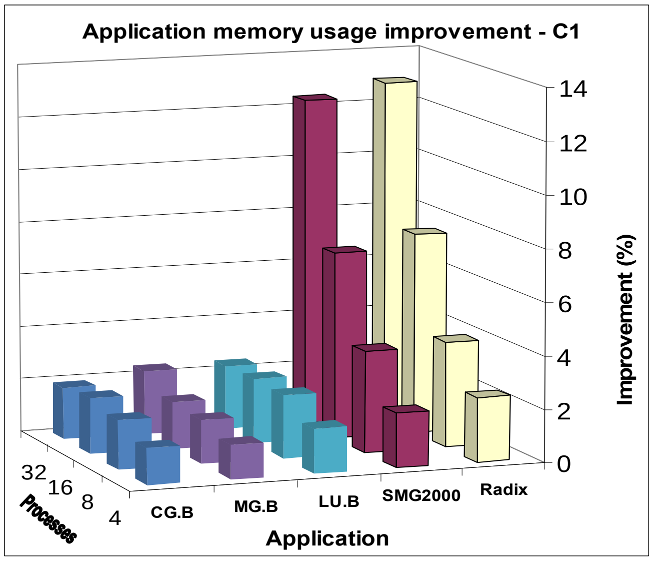
Socket buffers are the most significant contributors to the memory usage at the OS level. However, in many operating systems including Linux, the *Slab* allocation system [9] is used in which a pool of buffers are pre-allocated and assigned to the sockets when data is being communicated. This mechanism that is primarily used to alleviate the memory fragmentation effects hides the contribution of socket buffer sizes to the overall application memory usage. Therefore, the socket buffer allocation is not reflected in the total memory of the system, unless the pre-allocated slab buffers are filled and new buffers are reallocated due to high instantaneous network usage.

At the MPI layer, MVAPICH pre-allocates a number of general buffer pools with different sizes for each process. For the datagram QP that is established in both connection- and datagram-based modes, a number of buffers are picked from these pools and pre-posted as receive buffers to the QP. Once a new connection-based QP is established, a default number of 95 receive buffers are picked from the pools and posted to the QP. With a default size of 8KB for each buffer, a rough estimate of 800KB or 200 pages of 4KB size are required per connection for each process.

To measure the memory usage for the software-based iWARP, the total number of memory pages allocated to each MPI job in Linux (reported by the Linux proc file system) is used. This obviously excludes the OS-level memory usage, which cannot be directly measured due to the slab allocation system. The size of each memory page on our system is 4KB.

Figure 6.8 shows the improvement percentage for datagram-iWARP over connection-oriented iWARP. Table 6.1 provides the actual memory usage of the applications for different job sizes on C2 cluster. As observed, the overall memory usage of MPI applications can benefit from using datagrams. The benefit for some applications in NPB (such as CG) is relatively low and does not scale very well with the number of processes. This is primarily because each process in such applications usually communicates with a few partners. Therefore, the number of connections for each process will not scale by O(n^2) of the number of job processes. This means that the memory benefit can decrease or stay at the same level.

The case for SMG2000 and Radix is different from NAS applications. In these applications a process may communicate with all of the other processes, and therefore the number of dynamically allocated connections will increase by O(n^2) of the number of processes in the job. This is why we see significant increase in memory saving when the scale of the MPI job increases.



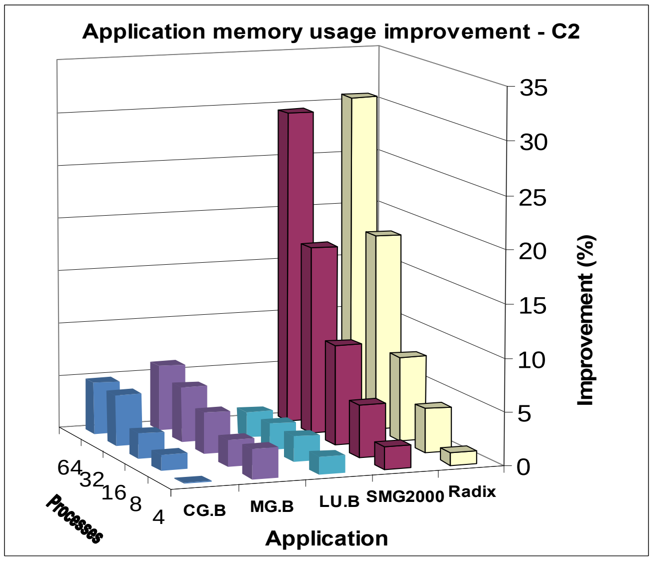


Figure 6.8 Memory usage improvement and scalability trend.

An obvious observation in both performance and memory usage benchmarks is that the C2 cluster results are significantly better than that of the C1 cluster. As discussed in Section 6.5, with the same number of processes, the inter-node communication between the nodes is quadrupled in C2. This also translates in more inter-process connections, which implies more memory savings on C2. The results lead to this conclusion that when the amount of inter-node communication rises, so do the benefits of using datagram-iWARP.

Table 6.1 Memory usage of the applications using datagram and connection modes on C2 cluster

	Memory pages (4 procs)		**Memory pages (16 procs)**		**Memory pages (64 procs)**	
Application	**UD**	**RC**	**UD**	**RC**	**UD**	**RC**
CG.B	234,924	235,204	556,801	570,207	1,994,428	2,097,580
MG.B	216,591	222,862	534,255	555,854	1,913,295	2,038,179
LU.B	153,453	155,982	473,740	495,422	1,894,628	1,978,663
SMG2000	102,899	105,138	418,063	461,478	1,815,192	2,577,276
RADIX	133,678	135,297	453,124	491,937	1,727,232	2,493,291

6.7 Summary

In this chapter, I discussed some challenges facing the current iWARP standard and devised a set of extensions to the standard to address these challenges. The proposed extensions enable iWARP to support the connection-less UDP protocol, primarily to overcome memory-usage and performance scalability issues of the connection-oriented TCP.

I also presented a software implementation of iWARP over UD to assess its potential benefits for HPC applications. The experimental results reveal that datagram-enabled iWARP increases the scalability of large-scale HPC applications. The solution also improves application performance due to using a lightweight transport protocol. In addition, verbs-level micro-benchmark results clearly show the potential benefits that other kinds of applications, such as datacenter services can receive from datagram-iWARP. The datagram-iWARP solution is also expected to reduce fabrication cost and support broadcast/multi-cast operations, once designed and implemented in hardware.

Chapter 7

Concluding Remarks and Future Work

Modern high-performance clusters are equipped with high-speed cluster interconnects. These interconnects offer advanced communication features such as RDMA and OS bypass, leading to high communication bandwidth and low latency. The clusters and especially their communication libraries need to efficiently utilize these features in order to translate the superior network performance into application performance. However, there are cases that the communication libraries are not able to provide efficient utilization of the underlying technology due to inefficiencies existing at different levels. For example, communication overhead in the MPI library results in higher data-transfer latency. In addition, in some cases, the library is not able to provide independent communication progress and as a consequence, the communication cannot be efficiently overlapped with computation. The memory and performance scalability of MPI over iWARP Ethernet is another problem. Due to the pervasiveness of Ethernet in clusters, it is important to address this issue in order to increase the memory and performance scalability of iWARP Ethernet and consequently the clusters built using it.

This dissertation is focused on improving the performance and scalability of inter-process communication in modern HPC clusters. The research is concentrated on the characteristics of modern high-performance interconnects, namely InfiniBand, iWARP Ethernet, and Myrinet, and their messaging libraries. The main contributions of this dissertation are:

(1) Evaluating the efficiency and shortcomings of the messaging layer in modern interconnects:

I have provided a comprehensive and in-depth analysis of the networks' communication characteristics in Chapter 3. Basic communication metrics such as latency and bandwidth, the scalability of the networks and libraries when using multiple independent connections on multi-core systems, the effect of memory registration and reusing memory buffers on the communication performance, and the ability of the libraries and networks in making independent communication progress and hiding communication latency are the main features assessed in Chapter 3. The results of this chapter point to:

- Inefficiency of the MPI library in utilizing the offloaded communication stack and making independent progress in large-message transfers over RDMA-enabled Interconnects, and consequently missing some of the communication/computation overlap opportunities in HPC applications.
- Inefficiency of the MPI small-message data transfer protocols due to buffer copies and memory registration required at both sides.
- Potential scalability issues for HPC applications running on iWARP Ethernet and InfiniBand networks due to multiple-connection scalability issues when using connection-oriented transports on large many-core systems.

The results in this chapter confirm that there is continuous demand to improve the performance and scalability of the message-passing library (i.e. MPI) on top of high-performance interconnects.

(2) Increasing the independent progress and overlap ability for large data transfers in MPI:

In Chapter 4, I have proposed a method that speculatively starts the communication at the receiver side of an MPI Rendezvous protocol, should the sender arrive later. This is instead of the current practice of waiting for the sender to start the Rendezvous communication, which would lead to poor communication progress. The results show that full independent progress and more than 80% overlap can be achieved when the receiver arrives earlier at a non-blocking call. The method is beneficial for decreasing the application communication time by up to 30%. The method is made adaptive such that it can stop the speculation if the application does not benefit from it.

(3) Reducing the communication latency for MPI small-message transfer protocols, both for two-sided and one-sided operations:

This has been achieved by removing the sender-side buffer copy (used to transfer the application data to a pre-registered memory buffer for RDMA communication), if the application buffer is reused frequently. In Chapter 5, I have shown that reusing application data buffers is a common practice in HPC applications. I have also shown that if an application buffer is reused a certain number of times, registering the buffer and directly using it for RDMA data transfer eliminates the overhead of an initial buffer copy, resulting in lower communication latency. I have also devised an adaptation mechanism to stop this practice if the application is not reusing its buffers as frequently as needed. This method has shown to be beneficial in decreasing the small-message transfer latency by up to 20% in MPI micro-benchmarks. It has also shown to reduce the communication time and runtime of MPI applications by up to 18%.

(4) Increasing the scalability and performance of HPC applications by extending the iWARP Ethernet standard to datagrams:

In Chapter 6, an extension of the current TCP-based iWARP protocol to the datagram domain (UDP protocol) has been proposed in order to equip Ethernet-based clusters with a scalable RDMA-enabled standard. This extension enables processes in large-scale HPC applications to communicate with a large number of processes without requiring a large number of connections that consume memory. The communication is also less complex, leading to lower cost of deployment and higher performance in relatively error-free HPC environments. The outcome of this work is a software-based datagram-iWARP. MPI has also been adapted to work on top of it. The results show considerable performance improvements, up to 30% latency reduction and up to 40% bandwidth improvement for micro-benchmarks as well as significant memory usage reduction (up to 30%) and application runtime reduction (up to 45%).

7.1 Future Work

For my future research, I plan to extend some of the directions that I have worked on during my PhD, as well as some new topics along the direction of improving communication performance and scalability in clusters. In the following, I will outline these directions.

7.1.1 Frequently-used Buffer Registration

The method proposed in Chapter 5 can be extended to domains other than HPC applications. SDP is one candidate for utilizing this method in its buffered-copy mode. The buffered-copy mode of the SDP protocol, which is mostly used for small-message transfers, can be equipped with this method to avoid part of the buffer copy overhead. File transfer protocols such as the Parallel Virtual File System (PVFS) and the Network File System (NFS-RDMA) are other candidates for utilizing this method.

7.1.2 iWARP Protocol Extension

The work presented in Chapter 6 is only a first step in standardization of datagram iWARP and can be continued in a number of directions.

I. Implementation of reliable datagram (RD) transport by using a reliable version of UDP protocol, and evaluation of MPI applications when reliability is provided by the iWARP rather than the MPI itself.

II. Extension of the current proposal to the tagged model and specifically, the definition and implementation of one-sided RDMA Read and Write operations on top of UD and/or RD transports.

III. Preparation of a socket interface on top of datagram iWARP to take advantage of it in socket-based datacenter applications. This can be an alternative to the current SDP protocol that only works on connection-oriented transports.

IV. Integration of the proposed extensions into the iWARP standard.

In particular, I am planning to work on the extension of the proposed datagram iWARP protocol to the tagged model[1]. This can be the first natural step towards an extended proposal for datagram iWARP.

7.1.3 MPI Topology Awareness

The topology functionality of the MPI-2 standard [63] provides a useful tool for mapping MPI processes to appropriate cores on the available nodes of a cluster. This functionality allows the user to construct a virtual topology based on the communication pattern of the application. It also allows the MPI implementation to efficiently map the user-defined virtual topology to the underlying physical topology of the cluster. In order to achieve the best communication performance, the MPI implementation should map as many communicating processes as possible to be closer together, in terms of communication cost.

Efficiently mapping the processes requires (1) using appropriate topology-mapping algorithms, (2) knowing the application communication pattern, and (3) appropriately detecting the underlying communication platform and its communication characteristics. In this direction, I plan to work on increasing the performance of MPI applications by implementing and using MPI topology functions, especially the distributed topology functions that have recently been added to the standard. These functions are key for scalable communication on future large-scale clusters, and will soon replace the use of centralized topology functions that are currently used in applications.

7.1.4 Quality of Service in MPI

I plan to extend my initial collaborative work on InfiniBand Quality of Service (QoS) [29] by exploiting IB QoS features in MPI communication. In [29] we have shown the effect of QoS provisioning on socket-based communication using IPoIB or SDP. The intention of my future work is to examine the effect of using InfiniBand QoS in MPI UD-based communication to differentiate between small and large messages, in order to provide a faster service to small and control messages in different MPI protocols.

[1] This work is currently being conducted. The work is led by Ryan Grant in collaboration with me and other colleagues.

References

[1] G. Amerson and A. Apon, "Implementation and Design Analysis of a Network Messaging Module using Virtual Interface Architecture," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing (CLUSTER '04),* 2004. pp. 255-265.

[2] L. Arber and S. Pakin, "The Impact of Message-Buffer Alignment on Communication Performance," *Parallel Processing Letters,* vol. 15, pp. 49-65. March.2005.

[3] Argonne National Laboratory.: http://www.anl.gov.

[4] Argonne National Laboratory. MPICH2 MPI Implementation.: http://www-unix.mcs.anl.gov/mpi/mpich2/.

[5] P. Balaji, S. Bhagvat, R. Thakur and D. K. Panda, "Sockets Direct Protocol for Hybrid Network Stacks: A Case Study with iWARP Over 10G Ethernet," in *Proceedings of the IEEE International Conference on High Performance Computing (HiPC'08),* Bangalore, India, 2008. pp. 478-490.

[6] P. Balaji, W. Feng, S. Bhagvat, D. K. Panda, R. Thakur and W. Gropp, "Analyzing the Impact of Supporting Out-of-Order Communication on in-Order Performance with iWARP," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07),* Reno, Nevada, 2007. pp. 1-12.

[7] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome and K. Yelick, "An Evaluation of Current High-Performance Networks," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS'03),* 2003. pp. 28a.

[8] A. V. Bhatt, "Creating a Third Generation I/O Interconnect," Intel® Developer Network for PCI Express* Architecture.2004.

[9] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator," in *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference (USTC'94),* Boston, Massachusetts, 1994. pp. 6.

[10] R. Brightwell, D. Doerfler and K. D. Underwood, "A Comparison of 4X InfiniBand and Quadrics ELAN-4 Technologies," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing (CLUSTER '04),* San Diego, USA, 2004. pp. 193-204.

[11] R. Brightwell, R. Riesen and K. D. Underwood, "Analyzing the Impact of Overlap, Offload, and Independent Progress for Message Passing Interface Applications," *International Journal of High Performance Computing Applications,* vol. 19, pp. 103-117. 2005.

[12] P. N. Brown, R. D. Falgout and J. E. Jones, "Semicoarsening Multigrid on Distributed Memory Machines," *SIAM Journal of Scientific Computing,* vol. 21, pp. 1823-1834. 2000.

[13] D. Buntinas, G. Mercier and W. Gropp, "Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem," in *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06),* Singapore, 2006. pp. 521-530.

[14] L. Chai, A. Hartono and D. K. Panda, "Designing High Performance and Scalable MPI Intra-Node Communication Support for Clusters," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'06),* 2006. pp. 1-10.

[15] T. Chen, R. Raghavan, J. N. Dale and E. Iwata, "Cell Broadband Engine Architecture and its First Implementation: A Performance View," *IBM Journal of Research and Development,* vol. 51, pp. 559-572. 2007.

[16] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman and P. Grun, "Remote Direct Memory Access Over the Converged Enhanced Ethernet Fabric: Evaluating the Options," in *Proceedings of the 2009 17th IEEE Symposium on High Performance Interconnects (HOTI '09),* 2009. pp. 123-130.

[17] P. Culley, U. Elzur, R. Recio and S. Bailey. Marker PDU Aligned Framing for TCP Specification (Version 1.0).RDMA Consortium. 2002.: http://www.rdmaconsortium.org/home/draft-culley-iwarp-mpa-v1.0.pdf.

[18] D. Dalessandro, A. Devulapalli and P. Wyckoff, "Design and Implementation of the iWARP Protocol in Software," in *Proceedings of Parallel and Distributed Computing and Systems,* 2005. pp. 471-478.

[19] D. Dalessandro and P. Wyckoff, "A Performance Analysis of the Ammasso RDMA Enabled Ethernet Adapter and its iWARP API," in *Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT 2005), Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'05),* 2005.

[20] D. Dalessandro, P. Wyckoff and G. Montry, "Initial Performance Evaluation of the NetEffect 10 Gigabit iWARP Adapter," in *Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT 2006), Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'06),* 2006. pp. 1-7.

[21] D. Dalessandro, A. Devulapalli and P. Wyckoff, "IWARP Protocol Kernel Space Software Implementation," in *6th Workshop on Communication Architectures for Clusters (CAC'06), Proceedings of the 2006 IEEE International Parallel and Distributed Processing Symposium (IPDPS'06),* Rhodes, Greece, 2006. pp. 1-8.

[22] D. Dalessandro and P. Wyckoff, "Memory Management Strategies for Data Serving with RDMA," in *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects (HOTI '07),* 2007. pp. 135-142.

[23] D. Doerfler and R. Brightwell, "Measuring MPI Send and Receive Overhead and Application Availability in High Performance Network Interfaces." in *Proceedings of Euro PVM/MPI Conference (PVM/MPI'06),* 2006. pp. 331-338.

[24] A. Faraj and X. Yuan, "Communication Characteristics in the NAS Parallel Benchmarks." in *Proceedings of Parallel and Distributed Computing Symposium (PDCS'02),* 2002. pp. 724-729.

[25] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan and D. K. Panda, "Performance Characterization of a 10-Gigabit Ethernet TOE," in *Proceedings of the 13th IEEE Symposium on High Performance Interconnects (HOTI '05),* 2005. pp. 58-63.

[26] B. Goglin, "Decoupling Memory Pinning from the Application with Overlapped on-Demand Pinning and MMU Notifiers," in *9th Workshop on Communication Architecture for Clusters (CAC'09), Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS'09),* Rome, Italy, 2009. pp. 1-8.

[27] B. Goglin, "Design and Implementation of Open-MX: High-Performance Message Passing Over Generic Ethernet Hardware," in *8th Workshop on Communication Architecture for Clusters (CAC'08), Proceedings of the 2008 IEEE International Parallel and Distributed Processing Symposium,* 2008. pp. 1-7.

[28] A. Grama, A. Gupta, G. Karypis and V. Kumar, *An Introduction to Parallel Computing: Design and Analysis of Algorithms.* MA, USA: Addison-Wesley Publishing, 2003.

[29] R. E. Grant, M. J. Rashti and A. Afsahi, "An Analysis of QoS Provisioning for Sockets Direct Protocol Vs. IPoIB Over Modern InfiniBand Networks," in *First International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), Proceedings of the 2008 International Conference on Parallel Processing (ICPP'08),* Portland, OR, 2008. pp. 79-86.

[30] B. Hauser. IWARP Ethernet: Eliminating Overhead in Data Center Designs.NetEffect Inc. 2006.: http://www.analogzone.com/nett0403.pdf.

[31] J. Hilland, P. Culley, J. Pinkerton and R. Recio. RDMA Protocol Verbs Specification (Version 1.0).RDMA Consortium. 2003.: http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf.

[32] T. Hoefler, "MPI: A Message-Passing Interface Standard -- Working-Draft for Nonblocking Collective Operations," MPI Forum.Apr. 2009.

[33] W. Huang, G. Santhanaraman, H. Jin and D. K. Panda, "Design Alternatives and Performance Trade-Offs for Implementing MPI-2 Over InfiniBand." in *Proceedings of Euro PVM/MPI Conference (PVM/MPI'05),* Sorrento, Italy, 2005. pp. 191-199.

[34] J. Hurwitz and W. Feng, "Analyzing MPI Performance Over 10-Gigabit Ethernet," *Journal of Parallel and Distributed Computing,* vol. 65, pp. 1253-1260. 2005.

[35] G. Huston. 2000. June 2000."TCP Performance,"*The Internet Protocol Journal,vol.3,(2),* :http://www.cisco.com/web/about/ac123/ac147/ac174/ac196/about_cisco_ipj_archive_article09186a00800c8417.html.

[36] HyperTransport Consortium. HyperTransport I/O Technology Overview.HyperTransport Consortium. 2004.: http://www.hypertransport.org/docs/wp/HT_Overview.pdf.

[37] IESP. International Exascale Software Project.: http://www.exascale.org/.

[38] IETF: The Internet Engineering Task Force. : http://www.ietf.org/.

[39] InfiniBand Trade Association. InfiniBand Architecture Specification.2007.: http://www.infinibandta.org/.

[40] Intel Corporation, "An Introduction to the Intel QuickPath Interconnect," Intel Corporation.Tech. Rep. 320412-001US, 2009.

[41] Intel Corporation. NetEffect NE010 10 Gb iWARP Ethernet Channel Adapter.: http://www.intel.com/Products/Server/Adapters/Server-Cluster/Server-Cluster-overview.htm.

[42] Intel Corporation. NetEffect NE020 10-Gb iWARP Ethernet Channel Adapter.2009.: http://www.intel.com/Assets/PDF/prodbrief/320918.pdf.

[43] H. W. Jin, S. Narravula, G. Brown, K. Vaidyanathan, P. Balaji and D. K. Panda, "Performance Evaluation of RDMA Over IP: A Case Study with the Ammasso Gigabit Ethernet NIC," in *Proceedings of the 14th International Symposium on High Performance Distributed Computing (HPDC'05),* Research Triangle Park, NC, 2005.

[44] M. Koop, T. Jones and D. K. Panda, "MVAPICH-Aptus: Scalable High-Performance Multi-Transport MPI Over InfiniBand," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'08),* Miami, FL, 2008. pp. 1-12.

[45] M. J. Koop, S. Sur, Q. Gao and D. K. Panda, "High Performance MPI Design using Unreliable Datagram for Ultra-Scale InfiniBand Clusters," in *Proceedings of the 21st Annual International Conference on Supercomputing (ICS '07),* Seattle, Washington, 2007. pp. 180-189.

[46] R. Kumar, A. R. Mamidala, M. J. Koop, G. Santhanaraman and D. K. Panda, "Lock-Free Asynchronous Rendezvous Design for MPI Point-to-Point Communication," in *Proceedings of the 15th European PVM/MPI Conference (PVM/MPI'08),* Dublin, Ireland, 2008. pp. 185-193.

[47] P. Lai, P. Balaji, R. Thakur and D. K. Panda, "ProOnE: A General-Purpose Protocol Onload Engine for Multi- and Many-Core Architectures," *Journal of Computer Science Research Development,* vol. 23, pp. 133-142. 2009.

[48] Lawrence Livermore National Laboratory. AMG 2006, ASC sequoia benchmarks code.: https://asc.llnl.gov/sequoia/benchmarks/.

[49] C. Lever. Linux Kernel Hash Table Behavior: Analysis and Improvements.Center for Information Technology Integration, University of Michigan. An Arbor, MI. 2000.: http://www.citi.umich.edu/techreports/reports/citi-tr-00-1.pdf.

[50] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff and D. K. Panda, "Performance Comparison of MPI Implementations Over InfiniBand, Myrinet and Quadrics," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC '03),* Phoenix, AZ, 2003. pp. 58.

[51] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, D. K. Panda and P. Wyckoff, "Microbenchmark Performance Comparison of High-Speed Cluster Interconnects," *IEEE Micro,* vol. 24, pp. 42-51. 2004.

[52] J. Liu, J. Wu and D. K. Panda, "High Performance RDMA-Based MPI Implementation Over infiniBand," *International Journal of Parallel Programming,* vol. 32, pp. 167-198. 2004.

[53] E. Lusk, "MPI-2: Standards Beyond the Message-Passing Model," in *Proceedings of the Conference on Massively Parallel Programming Models (MPPM '97),* 1997. pp. 43.

[54] K. Magoutis, "Memory Management Support for Multi-Programmed Remote Direct Memory Access (RDMA) Systems," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'05),* 2005. pp. 1-8.

[55] Mellanox Technologies. ConnectX Adapter Product Brief.Mellanox Technologies Inc. 2009.: http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX_VPI.pdf.

[56] Mellanox Technologies. Virtual Protocol Interconnect (VPI) Whitepaper.Mellanox Technologies Inc. 2009.: http://www.mellanox.com/pdf/prod_architecture/Virtual_Protocol_Interconnect_VPI.pdf.

[57] Mellanox Technologies Inc. : http://www.mellanox.com.

[58] B. Metzler, P. Frey and A. Trivedi. SoftRDMA, A Software Implementation of RDMAP/DDP/MPA.IBM Zurich Lab. 2010.: http://www.zurich.ibm.com/sys/servers/rdma_soft.html.

[59] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler and W. Rehm, "Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack," in *Proceedings of the Euro-Par Conference (Euro-Par'06),* Dresden, Germany, 2006. pp. 124-133.

[60] F. Mietke, R. Rex, T. Hoefler and W. Rehm, "Reducing the Impact of Memory Registration in InfiniBand," in *1St Kommunikation in Clusterrechnern Und Clusterverbundsystemen (KiCC),* Chemnitz, Germany, 2005.

[61] M. Morrow. Optimizing Memcpy Improves Speed.Embedded Systems Design. April.2004.: http://www.embedded.com/columns/technicalinsights/19205567?_requestid=189222.

[62] MPI Forum. : http://www.mpi-forum.org.

[63] MPI Forum, "MPI 2.2: A Message-Passing Interface Standard," MPI Forum.2009.

[64] Myricom Homepage. : http://www.myricom.com.

[65] Myricom Inc., "Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet," Myricom Inc.Tech. Rep. 1.0, 2006.

[66] S. Narravula, A. R. Mamidala, A. Vishnu, G. Santhanaraman and D. K. Panda, "High Performance MPI Over iWARP: Early Experiences," in *Proceedings of the International Conference on Parallel Processing (ICPP'07),* Xian, China, 2007.

[67] National Aeronautics and Space Administration (NASA). NAS parallel benchmarks (NPB) for MPI.: http://www.nas.nasa.gov/Resources/Software/npb.html.

[68] Network Based Computing lab. MVAPICH: MPI Over InfiniBand, 10GigE/iWARP and RoCE.:http://mvapich.cse.ohio-state.edu/.

[69] Network Working Group. Stream Control Transmission Protocol (SCTP).IETF. September.2007.: http://tools.ietf.org/html/rfc4960.

[70] Open MPI: Open Source High Performance Computing.: http://www.open-mpi.org/.

[71] OpenFabrics Alliance Homepage.: http://www.openfabrics.org/.

[72] L. Ou, X. He and J. Han, "An Efficient Design for Fast Memory Registration in RDMA," *Journal of Network Computing Applications,* vol. 32, pp. 642-651. 2009.

[73] S. Pakin, "Receiver-Initiated Message Passing Over RDMA Networks," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'08),* Miami, FL, 2008. pp. 1-12.

[74] Parallel Programming Laboratory. Charm++.: http://charm.cs.uiuc.edu/research/charm/.

[75] F. Petrini, E. Frachtenberg, A. Hoisie and S. Coll, "Performance Evaluation of the Quadrics Interconnection Network," *Journal of Cluster Computing,* vol. 6, pp. 125-142. 2003.

[76] Y. Qian, A. Afsahi and R. Zamani, "Myrinet Networks: A Performance Study," in *Third IEEE International Symposium on Proceedings of the Network Computing and Applications (NCA'04),* Cambridge, MA, 2004. pp. 323-328.

[77] M. J. Rashti and A. Afsahi, "Exploiting Application Buffer Reuse to Improve RDMA-Based MPI Communications," *Journal of Cluster Computing, Springer (Submitted),* 2010.

[78] M. J. Rashti and A. Afsahi, "Modern Interconnects for High-Performance Computing Clusters," in *Cluster Computing and Multi-Hop Network Research*, C. Jimenez and M. Ortego, Eds. Nova Science Publishers, Inc.2010.

[79] M. J. Rashti, R. E. Grant, P. Balaji and A. Afsahi, "IWARP Redefined: Scalable Connectionless Communication Over High-Speed Ethernet," in *17th IEEE International Conference on High Performance Computing (HiPC 2010) (Accepted for Publication),* Goa, India, 2010.

[80] M. J. Rashti and A. Afsahi, "10-Gigabit iWARP Ethernet: Comparative Performance Analysis with InfiniBand and Myrinet-10G," in *7th Workshop on Communication Architecture for Clusters (CAC'07), Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'07),* Long Beach, CA, 2007. pp. 1-8.

[81] M. J. Rashti and A. Afsahi, "A Speculative and Adaptive MPI Rendezvous Protocol Over RDMA-Enabled Interconnects," *International Journal of Parallel Programming,* vol. 37, pp. 223-246. 2009.

[82] M. J. Rashti and A. Afsahi, "Assessing the Ability of Computation/Communication Overlap and Communication Progress in Modern Interconnects," in *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects (HOTI '07),* Stanford, CA, 2007. pp. 117-124.

[83] M. J. Rashti and A. Afsahi, "Improving Communication Progress and Overlap in MPI Rendezvous Protocol Over RDMA-Enabled Interconnects," in *Proceedings of the 22nd International Symposium on High Performance Computing Systems and Applications (HPCS '08),* Quebec City, Canada, 2008. pp. 95-101.

[84] M. J. Rashti and A. Afsahi, "Improving RDMA-Based MPI Eager Protocol for Frequently-used Buffers," in *9th Workshop on Communication Architecture for Clusters (CAC'09),*

Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '09), Rome, Italy, 2009. pp. 1-8.

[85] RDMA Consortium. IWARP Protocol Specification.: http://www.rdmaconsortium.org.

[86] C. B. Reardon, A. D. George and C. T. Cole, "Comparative Performance Analysis of RDMA-Enhanced Ethernet," in *Workshop on High-Performance Interconnects for Distributed Computing (HPIDC'05),* 2005.

[87] R. Recio, P. Culley, D. Garcia and J. Hilland. An RDMA Protocol Specification (Version 1.0).RDMA Consortium. 2002.: http://www.rdmaconsortium.org/home/draft-recio-iwarp-rdmap-v1.0.pdf.

[88] J. C. Sancho, K. J. Barker, D. J. Kerbyson and K. Davis, "Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06),* Tampa, Florida, 2006. pp. 125.

[89] Sandia National Laboratories. LAMMPS molecular dynamics simulator.: http://lammps.sandia.gov/.

[90] H. Shah, J. Pinkerton, R. Recio and P. Culley. Direct Data Placement Over Reliable Transports (Version 1.0).RDMA Consortium. 2002.: http://www.rdmaconsortium.org/home/draft-shah-iwarp-ddp-v1.0.pdf.

[91] H. Shan, J. P. Singh, L. Oliker and R. Biswas, "Message Passing and Shared Address Space Parallelism on an SMP Cluster," *Journal of Parallel Computing,* vol. 29, pp. 167-186. 2003.

[92] A. G. Shet, P. Sadayappan, D. E. Bernholdt, J. Nieplocha and V. Tipparaju, "A Performance Instrumentation Framework to Characterize Computation-Communication Overlap in Message-Passing Systems," in *Proceedings of the 2006 IEEE International Conference on Cluster Computing (CLUSTER'06),* Barcelona, Spain, 2006. pp. 1-12.

[93] G. M. Shipman, T. S. Woodall, G. Bosilca, R. Graham and A. B. MacCabe, "High Performance RDMA Protocols in HPC," in *Proceedings of the 13th Euro PVM/MPI Conference (PVM/MPI'06),* Bonn, Germany, 2006. pp. 76-85.

[94] D. Sitsky and K. Hayashi, "An MPI Library which Uses Polling, Interrupts and Remote Copying for the Fujitsu AP1000+," in *Proceedings of the IEEE International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN'96),* Beijing, China, 1996.

[95] M. Small and X. Yuan, "Maximizing MPI Point-to-Point Communication Performance on RDMA-Enabled Clusters with Customized Protocols," in *Proceedings of the 23rd ACM International Conference on Supercomputing (ICS '09),* Yorktown Heights, NY, USA, 2009. pp. 306-315.

[96] Standard Performance Evaluation Corporation (SPEC). SPEC MPI2007.2009.Last Updated: 2010.

[97] H. Subramoni, P. Lai, M. Luo and D. K. Panda, "RDMA Over Ethernet - A Preliminary Study." in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'09),* New Orleans, LA, 2009. pp. 1-9.

[98] S. Sur, H. Jin, L. Chai and D. K. Panda, "RDMA Read Based Rendezvous Protocol for MPI Over InfiniBand: Design Alternatives and Benefits," in *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06),* New York, New York, USA, 2006. pp. 32-39.

[99] V. Tipparaju, G. Santhanaraman, J. Nieplocha and D. K. Panda, "Host-Assisted Zero-Copy Remote Memory Access Communication on InfiniBand," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'04),* Santa Fe, NM, 2004. pp. 31.

[100] TOP500 Supercomputer Sites.: http://www.top500.org.

[101] F. Trahay, A. Denis, O. Aumage and R. Namyst, "Improving Reactivity and Communication Overlap in MPI using a Generic I/O Manager," in *Proceedings of the 2007 Euro PVM/MPI Conference (PVM/MPI'07),* Paris, France, 2007. pp. 170-177.

[102] P. Wyckoff and J. Wu, "Memory Registration Caching Correctness," in *Proceedings of the IEEE International Conference on Cluster Computing and Grid (CCGrid'05),* Cardiff, UK, 2005. pp. 1008-1015.

[103] R. Zamani, Y. Qian and A. Afsahi, "An Evaluation of the Myrinet/GM2 Two-Port Networks," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04),* Tampa, FL, 2004. pp. 734-742.